CLIBRARY VERSION 3.2a

Library by: G.B.Gustafson
2243 South Belaire Drive
Salt Lake City, UT 84109
1-801-466-6820

Compiler: C/80, Version 2.0/3.0/3.1,
by The Software Toolworks

Assembler: M80.COM, by Microsoft

Linker: L80.COM, by Microsoft

Librarian: LIB.COM, by Microsoft

Page	Cast	Function	Description
1	void	_exit()	Abort exit, no flush or close.
1	void	a_exit()	Abort exit.
2	int	abort(code)	Abort exit to DDT
4	int	abs(x)	Integer absolute value.
5	int	access(name,mode)	Access file boolean.
7	float	acos(x)	Arc Cosine.
8	float	acot(x)	Arc Cotangent.
9	float	acsc(x)	Arc Cosecant.
10	float	asec(x)	Arc Secant.
11	float	asin(x)	Arc Sine.
12	float	atan(x)	Arc tangent.
13	float	atan2(x,y)	Arc tangent of y/x.
14	float	atof(s)	Ascii to float.
15	int	atoi(s)	Ascii to integer.
16	long	atol(s)	Ascii to long.
17	int	bdos(code,DE)	BDOS interface.
19	int	binary(x,v,n)	Binary search.
23	int	bios(code,BC)	BIOS interface.
24	int	brk(address)	Program break,
26	char*	bsearch(key,base,n,w,cmp)	
62	void	C_cmln()	Move C command line.
28	char*	calloc(m,s)	Alloc with null fill.
34	char*	Cbuf	Console buffer pointer.
34	char*	CC_CCP	Stack pointer for caller of CCHAIN.
30	int	ccBDOS(DE,BC)	BDOS interface, low level.
32	int	ccBIOS(DE,BC,f)	BIOS interface, low level.
33	void	ccmain()	Main startup code.
35	char	CCtlCk	Loop variable, terminal input.
36	float	ceil(f)	Ceiling, round up next integer.
34	void	Cexit ()	Flush and close re-direction.
103	int	cfree(a)	Free address a from calloc().
38	unsigned	cfs(file)	Compute file size.
39	void	chain(tpa,program)	Chain to load address and run.
41	void	chdir(s)	Change default drive and user.
42	int	chmod(name,mode)	Change file attributes.
44	void	CHmode(n)	Change console mode.
46	int	chown(nm,own,gr)	Change file owner and group.
47	int	clearerr(fp)	Clear error on stream fp.
48	int	close(fd)	Close file primitive.
21	char	Cmode	Console echo=0, line=1, noecho=2.
49	int	cmpf(n,m)	Compare floats n,m.
49	int	cmpi(n,m)	Compare integers n,m.
49	int	cmpl(n,m)	Compare longs n,m.
49	int	cmps(n,m)	Compare strings n,m.
51	int	comp(str1,str2)	Compare, see instr().
52	int	con(c)	Single char console output.
52	int	conout(c1,c2,...)	Multiple char console output.
53	int	Copen ()	< & > redirection file opener.
54	char*	core(n)	Alloc memory from heap.
55	float	cos(x)	Cosine.

Page	Cast	Function	Description
56	float	cosh(x)	Hyperbolic cosine.
57	int	creat(name,pmode)	Ascii create file.
57	int	creata(name,pmode)	Same as creat().
57	int	creatb(name,pmode)	Binary create file.
58	char*	CtlB	^B vector address data.
58	void	CtlB ()	^B processor, language C rules.
60	int	CtlCk()	Ctrl-C and Ctrl-B check.
61	char*	cvupper(s)	Convert string to upper case.
63	char*	decodF(fcb)	Decode file control block.
64	float	deg(x)	Conversion to degrees.
65	void	dsort(table)	Distribution sort.
67	void	dsort16(table)	Distribution sort, 16-bit numbers.
69	char*	edata,end,etext	System locations.
70	int	errno	Error return variable.
71	void	error(s1,s2,...)	Print strings to console.
72	void	exit()	Standard C exit. [29]
73	float	exp(x)	Power base e = 2.718 exponent x.
77	float	fabs(value)	Float Absolute value.
78	int	fclose(fp)	Flush and close stream.
82	int	fflush(fp)	Flush stream buffer to disk.
83	char*	fgets(s,n,fp)	Get a line from a stream.
84	int	fileno(fp)	Convert stream to descriptor.
85	char*	fillchr(dest,c,n)	Fill buffer with byte c.
86	int	fin	File descriptor for stream stdin.
87	int	fin ()	Function to return fin.
88	char*	findFIRST(e,f)	Find file f, extent e.
89	char*	findNEXT()	Find next after findFIRST().
90	void	fixfile(f,p)	Fix file f with pattern p.
91	float	floor(f)	Round down to integer.
92	float	fmod(x,y)	Remainder of x/y.
93	char*	fnb(s)	Find next blank or end of string.
94	FILE*	fopen (file,mode)	Standard C/80 file open.
96	FILE*	fopenU(file,mode)	Standard Unix file open.
96	FILE*	fopena(name,mode)	ASCII standard file open.
98	FILE*	fopenb(name,mode)	BINARY standard file open.
86	int	fout	File descriptor, stream stdout.
86	int	fout ()	Function to return fout.
100	void	fprintf(fp,control,arg1,arg2,...)	
101	int	fputs(s,fp)	Print string s to stream fp.
102	int	fread(buff,s,n,fp)	Read n packets of size s.
103	int	free(a)	Free address a from malloc().
104	FILE*	freopa(nm,mode,fp)	Same as freopen().
105	FILE*	freopb(nm,mode,fp)	BINARY reopen file nm in new mode.
104	FILE*	freopen(nm,mode,fp)	ASCII reopen file nm in new mode.
107	float	frexp(x,nptr)	Break x into mantissa and exponent.
108	int	fscanf(fp,control,arg1,arg2,...)	
110	int	fseek(fp,off,pos)	Position file, long offset.
112	int	ftell(fp)	File position report, C/80.
113	long	ftell(fp)	File position, Unix.
112	int	ftellr(fp)	File position report, C/80.

Page	Cast	Function	Description
114	void	ftoa(how,pr,f,s)	Float to ASCII conversions.
115	int	fwrite(buff,s,n,fp)	Write n packets size s to stream.
21	char	GC BIN	Binary flag byte.
116	int	getatt(name)	CP/M coded attribute.
118	int	getbyte()	Get a char, no echo.
119	int	getc(fp)	Get a stream char ASCII.
120	int	getcbinary(fp)	Get a stream char BINARY.
121	int	getchar()	Get a stdin char.
122	int	getddir(s,p,n)	Get directory file names & sizes.
126	unsigned	getdfree(drv,dfp)	Get disk free space & info.
129	long	getl(fp)	Get stream long integer BINARY.
130	int	getline(s,n)	Read line, include newline.
131	int	getln(s,n)	Read line, drop newline.
133	char*	getpass(q)	Read in password noecho, device.
134	int	getpid()	Get process ID. Stub.
135	char*	gets(s)	Read a line, drop newline.
136	unsigned	getw(fp)	Read stream 16-bit word BINARY.
137	void	heaps(base,n,cmp)	Heap sort.
139	char*	highmem()	High memory address return.
140	float	Horner(x,p,n)	Horner's polynomial evaluation.
141	char*	index(str,c)	Report position of c in str.
142	int	inport(port)	Fetch byte from CPU port.
143	void	insert(tail)	String insert to CP/M buffers.
144	int	instr(n,str1,str2)	Find substring match position.
145	char	IObin[MAXCHN]	Binary flag '\0' or 'b'.
145	char*	IObuf[MAXCHN]	Addresses of the file buffers.
145	char	IOch[MAXCHN]	Channel numbers (-1 means empty).
145	char*	IOdev	Device string 'con:rdr:pun:lst:'
145	int	IOend[MAXCHN]	Saves EOF record number for seek.
145	char*	IOfcb[MAXCHN]	File control block addresses.
145	int	IOind[MAXCHN]	Offset to next file buffer char.
145	char	IOmode[MAXCHN]	Mode byte 'r', 'w', 'u'.
145	int	IOnchx[MAXCHN]	Amount read by ffb() into buffer.
145	char	IOpchan[4]	Device numbers 252,253,254,255.
145	char	IOpeof[4]	Device EOF chars 26,26,26,12.
145	char	IOpread[4]	CP/M read functions 1,3,0,0.
145	char	IOpwrit[4]	CP/M write functions 2,0,4,5.
145	char	IOrw[MAXCHN]	0 or 1 for last read or write.
145	int	IOsect[MAXCHN]	Sector count in 128-byte hunks.
145	int	IOsize[MAXCHN]	Buffer size (256 default).
145	int	IOtmp	Storage, work variable.
150	int	isalnum(c)	Tests c for alphabetic or numeric.
150	int	isalpha(c)	Tests c for alphabetic.
150	int	isascii(c)	Tests c for range 0 to 127 decimal.
148	int	isatty(fd)	Test for console device.
150	int	iscntrl(c)	Test 0-31 decimal.
150	int	isdigit(c)	Tests for a valid digit, 0-9.
149	int	isgraph(c)	Test for graphics char.
150	int	islower(c)	-1 for 'a' <= c <= 'z', else 0.
150	int	isprint(c)	Test for a printable character.

Page	Cast	Function	Description
150	int	ispunct(c)	Test not ctrl nor alphanumeric.
150	int	isspace(c)	Test blank, tab, CR, LF.
150	int	isupper(c)	-1 for uppercase c, else 0.
150	int	isxdigit(c)	Test for hex digit, 0-9,A-F.
152	char*	itoa(x,s)	Integer to ASCII.
153	int	keystat()	Key and console buffer status.
154	long	labs(value)	Long Integer Absolute value.
155	float	ldexp(x,n)	Load new exponent into x.
156	float	ln(x)	Log base e, macro in math.h.
156	float	log(x)	Log base e, same as ln(x).
157	float	log10(x)	Log base 10, calculator log(x).
158	int	logged()	Return currently logged disk.
159	unsigned	loginv()	Return 16-bit disk log vector.
237	int	longjmp(env,val)	Software far goto, see setjmp.
160	char*	lowmem()	Low memory address.
161	char*	lsearch(key,base,n,w,cmp)	Linear search & insert.
163	char*	ltoa(x,s)	Long to ASCII.
164	void	makeFCB(fcb,name)	Make file control block from name.
165	char*	malloc(m)	Memory alloc.
65	struct	mathsort	Structure used in DSORT,DSORT16.
167	int	max(x,y)	Integer max.
37	int	MAXCHN[]	1+n, n = maximum number of files.
168	unsigned	memsize()	Amount of available heap memory.
168	int	midstr(s1,s2)	Same as instr(), but 2 args.
171	int	min(x,y)	Integer minimum.
172	char*	mktemp(tmp)	Temporary file name generator.
173	float	modf(x,nptr)	Split float to mantissa & exp.
174	char*	moveMEM(dest,s,n)	Move n bytes no-ripple from s.
175	char*	movmem(s,dest,n)	Same as moveMEM, reverse args.
252	struct	mstr	MBASIC strings, used in ssort3().
177	int	numsort(n,table,s)	16-bit numerical sort.
179	int	open(name,access)	ASCII open, descriptor level.
179	int	opena(name,access)	ASCII open, descriptor level.
181	int	openb(name,access)	BINARY open, descriptor level.
183	int	outport(port,val)	Print byte to port.
34	char*	p_fin	Redirection input filename pointer.
34	char*	p_fout	Redirection output filename pointer.
21	char	PC_BIN	Binary putc() flag.
184	char	peekb(addr)	Fetch byte at address.
184	long	peekl(addr)	Fetch long integer at address.
184	unsigned	peekw(addr)	Fetch unsigned word at address.
185	int	perror(s)	Print last system errno error.
187	void	pokeb(addr,x)	Deposit byte at address.
187	void	pokel(addr,x)	Deposit long integer at address.
187	void	pokew(addr,x)	Deposit word at address.
188	float	pow(x,y)	Standard power x to the y.
189	float	pow10(y)	Standard power 10 to the y.
190	int	printf(control,arg1,arg2,...)	
195	int	putc(x,fp)	Print byte to stream ASCII.
197	int	putcbinary(x,fp)	Print byte to stream BINARY.

Page	Cast	Function	Description
199	int	putchar()	Print byte to stdout ASCII.
200	long	putl(x,fp)	Print BINARY long to stream.
201	int	puts(s)	Print string with newline.
202	unsigned	putw(n,fp)	Print BINARY word to stream.
203	int	qsort(base,n,s,CMP)	Quicksort, Unix standard.
205	void	quick(lo,hi,base,CMP)	Quicksort, alternate.
207	float	rad(x)	Radian value of argument x.
208	unsigned	rand()	Returns next random number.
208	unsigned	rand1()	Special service routine, rand.
209	int	rdDISK(track,record,drive,buffer)	Direct-disk read.
294	unsigned	read (fp,buf,n)	C/80 read binary, 128 records.
212	int	read(fd,buffer,n)	ASCII Unix read n bytes to buffer.
212	int	reada(fd,buffer,n)	ASCII read n bytes to buffer.
213	int	readb(fd,buffer,n)	BINARY read n bytes to buffer.
215	char*	realloc(ptr,s)	Re-allocate from malloc() call.
217	int	rename(old,new)	Rename a disk file.
218	void	reset()	Reset all drives.
219	char*	reverse(str)	Reverse string in place.
220	void	rewind(fp)	Position stream to start.
222	char*	rindex(str,c)	Reverse search string for char c.
223	void	run(cmd)	Run COM file with args.
224	int	sbinary(x,v,n)	Binary search string table.
225	char*	sbrk(n)	Basic heap allocate.
227	int	scanf(control,&arg1,&arg2,...)	
231	int	seek(fp,off,type)	Integer seek().
233	int	seekend(fd)	Seek to end of file.
234	void	seldsk(x)	Select disk by number.
235	int	setatt(name,code)	Set file attributes.
236	int	setbuf(fp,buffer)	Change file buffer location.
237	int	setjmp(env)	Set up far GOTO. See longjmp.
240	unsigned	sfree(n)	Free up space assigned by sbrk().
241	void	shells(p,n,CMP)	Shell-Metzner sort.
243	int	signal(sig,f)	Unix signal (interrupt).
244	float	sin(x)	Sine.
245	float	sinh(x)	Hyperbolic sine.
246	char*	sob(s)	Skip over blanks.
247	int	sprintf(s,control,arg1,arg2,...);	
248	float	sqrt(x)	Square root.
208	void	srand(seed)	Seeds the random number generator.
249	int	sscanf(s,control,arg1,arg2,...)	
251	void	ssort2(n,table,len)	Shell sort fixed length strings.
252	void	ssort3(size,&table)	Shell sort MBASIC string table.
253	#define	stderr (FILE *)252	
253	#define	stdin (FILE *)fin_()	
253	#define	stdout (FILE *)fout_()	
254	char*	strcat(str1,str2)	String concatentate.
255	char*	strchr(s,c)	Find c in string s.
256	int	strcmp(str1,str2)	Compare strings.
257	char*	strcpy(str1,str2)	Copy strings.
259	int	strcspn(s,set)	Find complement span from set.

Page	Cast	Function	Description
260	int	strlen(str)	String length.
261	char*	strncat(s1,s2,n)	Concatenate strings.
262	int	strncmp(s1,s2,n)	Compare strings.
263	char*	strncpy(s1,s2,n)	Copy strings /w NULL FILL.
264	char*	strpbrk(s,set)	Pointer span from set.
265	int	strpos(s,c)	Offset char c in string s.
266	char*	strrchr(s,c)	Pointer rev char c in string s.
267	char*	strrpbrk(s,set)	Pointer rev span from set.
268	int	strrpos(s,c)	Offset rev char c in string s.
269	int	strspn(s,set)	Offset span from set.
270	int	swab(s,d,n)	Swap words in memory.
271	int	system(s)	System call by $$$.SUB.
272	float	tan(x)	Tangent.
274	float	tanh(x)	Hyperbolic tangent
275	void	timer(n)	H89/Z100 timer.
276	int	toascii(x)	Convert to ASCII range 0-127.
277	int	toint(x)	Convert hex byte to integer.
278	int	tokens(table,tail)	Command line decoder. [29]
280	int	tolower(x)	Convert to lower case.
33	EQU	TOT_SZ	Total size of FCB & file buffers.
281	int	toupper(x)	Convert to upper case.
282	char*	ttyname(fd)	Name of console device.
82	int	uflush(fp)	Unix stream flush. See fflush.
284	int	ungetc(x,fp)	Push back char on stream.
285	EQU	unixio	All unix stub functions.
289	int	unlink(filename)	Erase directory entry.
290	char*	utoa(n,s)	Unsigned to ASCII base 10.
290	char*	utoab(n,s)	Unsigned to ASCII base 2.
290	char*	utoab(o,s)	Unsigned to ASCII base 8.
290	char*	utoax(n,s)	Unsigned to ASCII base 16.
291	void	waitz(n)	Time-out, H89/Z100.
292	int	wrDISK(track,record,drive,buffer)	Direct-disk write.
293	unsigned	write (fp,buf,n)	C/80 standard binary 128-write.
295	int	write(fd,buffer,n)	ASCII Unix write n bytes.
295	int	writea(fd,buffer,n)	ASCII write n bytes.
296	int	writeb(fd,buffer,n)	BINARY write n bytes.

Page	Cast	Function	Stub Return
285	int	acct(file)	Return 0.
285	unsigned	alarm(seconds)	Return 0.
285	char*	cuserid(s)	Return (char *)0.
285	int	dup(old)	Return -1 for error.
285	int	dup2(old,new)	Return -1 for error.
285	int	fork()	Return -1 for error.
285	int	fstat(fd,buf)	Return -1 for error.
285	void	ftime(tp)	Void return.
285	int	getpid()	Return 0.
285	int	getuid()	Return 0.
285	int	getgid()	Return 0.
286	int	geteuid()	Return 0.
286	int	getegid()	Return 0.
286	int	ioctl(fd,request,argp)	Return -1 for error.
286	int	getpgrp()	Return 0.
286	int	getppid()	Return 0.
286	int	kill(pid,sig)	Return -1 for error.
286	int	link(path1,path2)	Return -1 for error.
286	int	mknod(path,mode,dev)	Return -1 for error.
286	int	mount(spec,dir,rwflag)	Return -1 for error.
286	int	nice(incr)	Return 0.
286	void	pause()	Void return.
286	int	pclose(fp)	Return -1 for error.
286	int	pipe(fd)	Return -1 for error.
286	FILE*	popen(command,type)	Return 0.
286	void	profil(buff,bufsize,offset,scale)	Void return.
286	int	ptrace(request,pid,addr,data)	Return -1 for error.
286	int	putpwent(p,fp)	Return -1 for error.
286	int	setgid(gid)	Return -1 for error.
287	int	setpgrp()	Return 0.
287	int	setuid(uid)	Return -1 for error.
287	int	stat(name,buf)	Return -1 for error.
287	void	sync()	Void return.
287	long	time(tloc)	Returns (long)0
287	long	times(buffer)	Return -1 for error.
287	int	stime(tp)	Return -1 for error.
287	int	umask(cmask)	Return 0.
287	int	umount(spec)	Return -1 for error.
287	int	uname(name)	Return 0.
287	char*	userid(s)	Return (char *)0
287	int	ustat(dev,buf)	Return -1 for error.
287	int	utime(path,times)	Return -1 for error.
287	int	wait(stat_loc)	Return -1 for error.

Page	Cast	Function	Stub Return
288	struct	group *getgrent()	Return 0.
288	struct	group *getgrgid(gid)	Return 0.
288	struct	group *getgrnam(name)	Return 0.
288	int	setgrent()	Return 0.
288	int	endgrent()	Return 0.
288	char*	getlogin()	Return (char *)0.
288	int	getpw(uid,buf)	Return -1 for error.
288	struct	passwd *getpwuid(uid)	Return 0.
288	struct	passwd *getpwent()	Return 0.
288	struct	passwd *getpwnam(name)	Return 0.
288	int	setpwent()	Return 0.
288	int	endpwent()	Return 0.
288	int	monitor(lowpc,highpc,buffer,bufsize,nfunc)	Return 0.
288	int	sleep(seconds)	Return 0.

Module GETBYT — Special get byte no echo
Program 15, Data area 0
Entries: GETBYT
Externals: CMODE, GETCHA

Module STRCMP — K&R string compare
Program 28, Data area 0
Entries: STRCMP
Externals: RN$, S.1

Module STRCPY — K&R string copy
Program 21, Data area 0
Entries: STRCPY

Module STRCAT — K&R string concatenate
Program 28, Data area 0
Entries: STRCAT

Module STRNCM — K&R string n-compare
Program 42, Data area 0
Entries: STRNCM
Externals: RN$, S.1

Module STRNCP — K&R string n-copy with null fill.
Program 36, Data area 2 — WARNING: May not terminate string.
Entries: STRNCP

Module STRNCA — K&R string n-concatenate
Program 45, Data area 2
Entries: STRNCA

Module STRCSP — Harbison & Steele strcspn()
Program 35, Data area 0
Entries: STRCSP

Module STRPBR — Harbison & Steele strpbrk()
Program 32, Data area 0
Entries: STRPBR

Module STRPOS — Harbison & Steele strpos()
Program 21, Data area 0
Entries: STRPOS
Externals: $RM

Module STRRPB — Harbison & Steele strrpbrk()
Program 42, Data area 0
Entries: STRRPB

Module STRRPO — Harbison & Steele strrpos()
Program 31, Data area 0
Entries: STRRPO
Externals: $RM

Module MIN — Integer minimum value
Program 16, Data area 0
Entries: MIN
Externals: C.LT

Module ATOI — ASCII text to integer 16-bit
Program 31, Data area 0
Entries: ATOI
Externals: $CVI, C.NEG, SOB

Module TOINT — Hex digit to integer byte
Program 34, Data area 0
Entries: TOINT

Module ITOA — Integer 16 bit to ASCII string
Program 34, Data area 0
Entries: ITOA
Externals: $NUMST

Module REVERS — Reverse string in place
Program 34, Data area 0
Entries: REVERS
Externals: HLCPDE

Module GETS — Get string from console
Program 17, Data area 0
Entries: GETS
Externals: GETLN

Module GETLN — Get string from console with max limit
Program 28, Data area 2
Entries: GETLN
Externals: GETLIN

Module GETLIN — Get line from console K&R.
Program 49, Data area 0
Entries: GETLIN
Externals: GETCHA, HLCPDE, RN$

Module FPUTS — Standard K&R fputs() to stream
Program 33, Data area 0
Entries: FPUTS
Externals: $RM, PUTC, RN$

Module FGETS — Standard K&R fgets().
Program 59, Data area 0
Entries: FGETS
Externals: GETC, HLCPDE, RN$

Module ISGRAP — Test for graphic character class
Program 16, Data area 0
Entries: ISGRAP

Module ISDIGI — Test for digit 0123456789
Program 17, Data area 0
Entries: $ISDIG, ISDIGI

Module ISSPAC — Test for space "\t\40\r\n"
Program 14, Data area 0
Entries: ISSPAC
Externals: ISSPC$

Module ISCNTR — Test for control character
Program 13, Data area 0
Entries: ISCNTR

Module ISPRIN — Test for printable character
Program 17, Data area 0
Entries: ISPRIN

Module ISPUNC — Test for punctuation
Program 41, Data area 0
Entries: ISPUNC

Module TOASCI — Convert to ASCII range 0-127
Program 11, Data area 0
Entries: TOASCI

Module TOLOWE — Convert to lower case letter
Program 17, Data area 0
Entries: TOLOWE

Module TOUPPE — Convert to upper case letter
Program 17, Data area 0
Entries: TOUPPE

Module CVUPPE — Convert whole string to upper case
Program 15, Data area 0
Entries: $CVUPP, CVUPPE
Externals: NAM.U

Module ABS — Integer absolute value
Program 11, Data area 0
Entries: ABS
Externals: C.NEG

Module MAX — Integer maximum value
Program 16, Data area 0
Entries: MAX
Externals: C.LT

Module CCMAIN — Main stub required for each main program
Program 132, Data area 22
Entries: $LM, A_EXIT, CBUF, CC_CCP, CMODE, CTLB, CTLB., CTLB_EXIT, FIN, FOUT, P_FIN, P_FOUT
Externals: $END, CEXIT_, COPEN_, C_MCLN, EX_SPC, MAIN, TOKENS, TOT_SZ

Module ERRNO — Error number variable
Program 132, Data area 2
Entries: ERRNO

Module CMCLN — C command line copier.
Program 67, Data area 12
Entries: C_MCLN
Externals: CBUF

Module CORE — Gets memory from sbrk().
Program 39, Data area 0
Entries: CORE
Externals: EXIT, SBRK

Module FINFOU — Stdin and stdout returns.
Program 8, Data area 0
Entries: FIN_, FOUT_
Externals: FIN, FOUT

Module ISASCI — Test for ASCII character class
Program 13, Data area 0
Entries: ISASCI

Module ISALNU — Test for alphanumeric character class
Program 13, Data area 0
Entries: ISALNU
Externals: $ISALP, $ISDIG

Module ISALPH — Test for alphabetic character class
Program 13, Data area 0
Entries: $ISALP, ISALPH
Externals: $ISLOW, $ISUPP

Module ISUPPE — Test for upper case letter
Program 17, Data area 0
Entries: $ISUPP, ISUPPE

Module ISLOWE — Test for lower case letter
Program 17, Data area 0
Entries: $ISLOW, ISLOWE

Module ISXDIG — Test hex digit 0123456789ABCDEFabcdef
Program 18, Data area 0
Entries: ISXDIG
Externals: $ISDIG

Module CONOUT
Program 30, Data area 0
Entries: CONO 1, CONO 2
Externals: $CON, PROCL.
Console only character output with multiple arguments.

Module TYPE
Program 73, Data area 0
Entries: TYPE 1, TYPE 2, TYPE_3
Externals: PROCL. PUTCHA
Re-direction string output. Multiple arguments.

Module PRINTF
Program 31, Data area 0
Entries: PRTF 1, PRTF 2
Externals: PRNTF., PUTCHA
Printf() header, K&R hispeed with many features omitted. See also CLIBM printf.

Module FPRINT
Program 42, Data area 0
Entries: FPRT 1, FPRT 2
Externals: PRNTF., PUTC
K&R fprintf(), see printf() above.

Module SPRINT
Program 47, Data area 0
Entries: SPRT 1, SPRT_2
Externals: PRNTF.
K&R sprintf(), see printf() above.

Module PRNTF
Program 250, Data area 25
Entries: FMTLOC, PRNTF, PRNTF.
Externals: $CVI, $NUMST, $SETPA, H. HLCPDE, PROCL.
Main printf routine.

Module SETPAD
Program 79, Data area 0
Entries: $SETPA
Externals: HLCPDE, S.1
Pad character set routine for printf().

Module NUMSTR
Program 169, Data area 0
Entries: $NUMST
Externals: C.NEG
Number conversion for printf().

Module CVI
Program 29, Data area 0
Entries: $CVI
Integer conversion for printf().

Module PROCL
Program 29, Data area 0
Entries: PROCL.
Externals: HLCPDE
Process list of multiple arguments.

Module RESET
Program 6, Data area 0
Entries: RESET
Reset disk system

Module STRSPN Program 39, Data area 0 Entries: STRSPN	Harbison & Steele strspn()
Module INDEX Program 17, Data area 0 Entries: INDEX, STRCHR Externals: RN$	Unix-style index()
Module RINDEX Program 31, Data area 0 Entries: RINDEX Externals: RN$	Unix-style rindex()
Module INSTR Program 67, Data area 0 Entries: INSTR Externals: COMP$, RN$	Microsoft MBASIC-style instr()
Module MIDSTR Program 51, Data 0 Entries: MIDSTR Externals: RN$	Faster instr() for 2 arguments.
Module COMP Program 39, Data area 0 Entries: COMP COMP$ Externals: RM, RN	Server for instr() function above
Module STRLEN Program 16, Data area 0 Entries: STRLEN	K&R string length
Module FILLCH Program 23, Data area 0 Entries: FILLCH	PASCAL-style fill memory. Arguments like strncpy().
Module MOVMEM Program: 10, Data area 0 Entries: MOVMEM Externals: MOVEME	Memory move with reversed arg order.
Module MOVEME Program 53, Data area 0 Entries: MOVEME Externals: HLCPDE, MOVE$	Memory move without overlap faults, uses Z80 block move on Z80 CPU. Arguments are like strncpy().
Module ERROR Program 50, Data area 0 Entries: ERROR, ERR_1, ERR_2 Externals: $CON, PROCL.	Console-only string out. Mult-args.

Module PUTC — Put character ASCII mode, K&R
Program 135, Data area 0
Entries: PUTC
Externals: $PCH, $RM, FFB, FFLUSH, IOBIN, IOBUF, IOIND, IORW, IOSIZE, MAXCHN, PC_BIN

Module PCH — Subroutine for putc()
Program 40, Data area 0
Entries: $PCH
Externals: $PU, IOPWRI, PC_BIN

Module OR — Logical OR function
Program 10, Data area 0
Entries: O.

Module XOR — Exclusive OR function
Program 10, Data area 0
Entries: X.

Module AND — Logical AND function
Program 10, Data area 0
Entries: A.

Module HLDE — Test HL=DE
Program 14, Data area 0
Entries: N., N.1

Module CMPSIG — Compare signs, register level
Program 45, Data area 0
Entries: C.GE, C.GT, C.LE, C.LT

Module ASR — Arithmetic shift right
Program 14, Data area 0
Entries: C.ASR

Module USR — Unsigned shift right
Program 13, Data area 0
Entries: C.USR

Module ASL — Arithmetic shift left
Program 7, Data area 0
Entries: C.ASL

Module Q — Store HL at address on stack
Program 8, Data area 0
Entries: Q.

Module CMULT — Multiplication, signed
Program 28, Data area 0
Entries: C.MULT

Module PUTCHA K&R putchar()
Program 26, Data area 0
Entries: PUTCHA
Externals: $PCH, FOUT, PUTC

Module GETCHA K&R getchar()
Program 14, Data area 0
Entries: GETCHA
Externals: $B8, FIN, GETC

Module UNGETC K&R stream unget character
Program 119, Data area 0
Entries: UNGETC
Externals: $UNGET, FIN, IOCH, IOFCB, IOIND, IOMODE, IOSECT, IOSIZE, MAXCHN

Module GETCBI Get character binary
Program 49, Data area 0
Entries: GETCBI
Externals: CMODE, GC_BIN, GETC

Module GETC Get character ASCII mode
Program 165, Data area 1
Entries: GC_BIN, GETC
Externals: $GCH, $RM, FFB, FFLUSH, IOBIN, IOBUF, IOIND, IONCHX, IOSIZE, MAXCHN, READ

Module KEYSTA Keyboard and console buffer status
Program 19, Data area 0
Entries: KEYSTA
Externals: $KEYST, $RM, CBUF, CCNT

Module GETPUT Subroutine for device input
Program 120, Data area 2
Entries: $B8, $GCH, $RLIN, $UNGET, CCNT, CCTLCK

Externals: $BREAK, $CONIN, $PU, $RADD, $RM, CBUF, CMODE, CTLB., CTLCK, IOPREA,

Module SELDSK Select disk drive
Program 13, Data area 0
Entries: SELDSK

Module PUTCBI Put character binary
Program 26, Data area 0
Entries: PUTCBI
Externals: PC_BIN PUTC

Module CDIV — Division, signed
Program 72, Data area 0
Entries: C.DIV, C.UDV
Externals: C.NEG

Module SWITCH — Switch case subroutine
Program 26, Data area 0
Entries: .SWITC

Module CNEG — Compute HL = -HL
Program 8, Data area 0
Entries: C.COM, C.NEG

Module CNOT — Return TRUE if HL==0
Program 10, Data area 0
Entries: C.NOT
Externals: E.2

Module UGE — Unsigned >= test
Program 18, Data area 0
Entries: C.UGE, C.ULE

Module UGT — Unsigned > test
Program 14, Data area 0
Entries: C.UGT, C.ULT

Module MEMSIZ — Memory size for heap space managed by sbrk().
Program 25, Data area 0
Entries: HIGHME, LOWMEM, MEMSIZ
Externals: $LM, $RM, HLCPDE, S.1

Module CFS — Compute file size, virtual
Program 39, Data area 0
Entries: CFS
Externals: $RM, .XFCB, H.

Module RENAME — Rename a file to new name
Program 38, Data area 0
Entries: RENAME
Externals: .XFCB

Module UNLINK — Erase a disk file by name
Program 26, Data area 0
Entries: UNLINK
Externals: .XFCB

Module LOGGED — Report currently logged disk
Program 9, Data area 0
Entries: LOGGED

Module LOGINV — Return 16 bit vector of logged drives
Program 5, Data area 0
Entries: LOGINV

Module FINDDI
Program 58, Data area 0
Entries: FINDFI, FINDNE
Externals: $RM, .XFCB

Module FIXFIL — Fix a file name from template string
Program 102, Data area 0
Entries: FIXFIL
Externals: .XFCB, DECODF, MOVE$

Module DECODF — Decode a file control block to a file name.
Program 70, Data area 0
Entries: DECODF

Module CHMODE — Change to/from character mode
Program 9, Data area 0
Entries: CHMODE
Externals: CMODE

Module MAKEFC — Make a file control block
Program 12, Data area 0
Entries: MAKEFC
Externals: .XFCB

Module TOKENS — Token decoder for C command line
Program 132, Data area 0
Entries: ISQUO$, TOKEN$, TOKENS
Externals: ISSPC$, SOB$

Module COPEN — Re-direction file fin/fout open routine
Program 51, Data area 0
Entries: COPEN
Externals: A_EXIT, FOPEN

Module CEXIT — Re-direction file fclose() routine
Program 12, Data area 0
Entries: CEXIT
Externals: FCLOSE, FOUT

Module FOPEN — Standard C/80 fopen(), not K&R, but close.
Program 219, Data area 5
Entries: FOPEN, FOPEN
Externals: $PU, .CPCI, .XFCB, IOBIN, IOBUF, IOCH, IODEV, IOEND, IOFCB, IOMODE, IONCHX, IORW, IOSECT, IOSIZE, MAXCHN, RN$, SBRK

Module FCLOSE — Standard K&R fclose() for streams.
Program 73, Data area 0
Entries: FCLOSE
Externals: $PU, FFLUSH, IOCH, IOFCB, IOPEOF, IOPWRI, MAXCHN, RN$

Module REWIND — Rewind open stream
Program 42, Data area 0
Entries: REWIND
Externals: FFLUSH, IOFCB, IOIND, IOSECT

Module FFLUSH — Non-standard file flush - does not put
Program 133 , Data area 0 — FCB into directory. See CLIBU.REL.
Entries: FFLUSH
Externals: $RM, HLCPDE, IOBIN, IOBUF, IOFCB, IOIND, IOMODE, IORW, IOSECT, RN$, WRITE

Module FFB — File file buffer from disk
Program 96, Data area 0
Entries: FFB
Externals: IOBIN, IOBUF, IOIND, IONCHX, IORW, IOSIZE, READ

Module WRITE — C/80 write binary. See writea, writeb.
Program 186, Data area 30
Entries: READ, READ , WRITE, WRITE
Externals: CMODE, CTLCK, IOFCB, IOSECT, MAXCHN, RN$, S.1

Module CON — Console output, uses BDOS.
Program 9, Data area 0
Entries: CON

Module CONBUF — Subroutines for console buffer use.
Program 59, Data area 0 — See putc(), getc().
Entries: $BREAK, $RADD, CTLCK
Externals: A_EXIT, CBUF, CMODE, CTLB., $CON, $CONIN, $KEYST

Module CONIO — Console input & output routines
Program 40, Data area 0
Entries: $CON, $CONIN, $KEYST
Externals: CMODE

Module XFCB — Make file control block, low level
Program 112, Data area 0
Entries:
 .XFCB
Externals: ISSPC$, NAM.U

Module CPC1 — Subroutine for fopen
Program 36, Data area 0
Entries:
 .CPC1
Externals: IOFCB, RN$, SBRK

Module SFREE — Frees space set up by sbrk().
Program 108, Data area 0
Entries: SFREE
Externals: $END, $LM, H., HLCPDE, IOBUF, IOFCB, IOSIZE, MAXCHN, RN$, S.1

Module SBRK — Gives heap address for space request
Program 38, Data area 0 — or -1 on failure
Entries: SBRK
Externals: $LM, $RM, HLCPDE, S.1

Module BRK — Sets program break address, see Banahan
Program 23, Data area 0 — and Rutter and CP/M-68k manual.
Entries: BRK
Externals: $LM, $RM, HLCPDE

Module $PU — Physical unit subroutine
Program 22, Data area 0
Entries: $PU
Externals: $RM

Module IOTABL — Data for all file I/O and buffers.
Program 22, Data area 161
Entries: IOBIN, IOBUF, IOCH, IODEV, IOEND, IOFCB, IOIND, IOMODE, IONCHX, IOPCHA, IOPEOF, IOPREA, IOPWRI, IORW, IOSECT, IOSIZE, IOTMP, MAXCHN
Externals: $END, $OFF1, $OFF2, $OFF3, $OFF4, $OFF5, $OFF6, $SIZ1, $SIZ2, $SIZ3, $SIZ4, $SIZ5, $SIZ6, $TAG1, $TAG2, $TAG3, $TAG4, $TAG5, $TAG6

Module NAMU — Upper case name converter for files
Program 11, Data area 0
Entries: NAM.U

Module E — Test HL=0 in various ways
Program 25, Data area 0
Entries: E., E.0, E.1, E.2, E.F, E.T

Module H — Fetch to HL from address in HL.
Program 5, Data area 0 — Register level function.
Entries: H.

Module HLCPDE — Compare register HL to DE
Program 6, Data area 0
Entries: HLCPDE

Module SOB — Skip over blanks in a string
Program 13, Data area 0
Entries: SOB, SOB$
Externals: ISSPC$

Module FNB — Find next blank in a string
Program 15, Data area 0
Entries: FNB, FNB$
Externals: ISSPC$

Module ISSPC — Test for space characters, low level
Program 12, Data area 0
Entries: ISSPC$

Module PEEKB
Program 8, Data area 0
Entries: PEEKB
Peek at address for byte return

Module PEEKW
Program 9, Data area 0
Entries: PEEKW
Peek address for 16 bit return

Module POKEB
Program 8, Data area 0
Entries: POKEB
Poke byte at address

Module POKEW
Program 11, Data area 0
Entries: POKEW
Poke 16 bit word at address

Module BDOS
Program 8, Data area 0
Entries: BDOS
Externals: CCBDOS
Standard BDOS interface

Module BIOS
Program 12, Data area 0
Entries: BIOS
Externals: CCBIOS
Standard BIOS interface

Module CCBDOS
Program 40, Data area 0
Entries: CCBDOS
Special reverse argument BDOS interface with all registers.

Module CCBIOS
Program 52, Data area 0
Entries: CCBIOS
Special reverse argument BIOS interface with all registers and special returns, depending on function number.

Module SETJMP
Program 38, Data area 0
Entries: LONGJM, SETJMP
Set jump companion for long jump return

Module MOVE
Program 25, Data area 0
Entries: MOVE$
Low level memory move

Module SUBT16
Program 10, Data area 0
Entries: S., S.1
16 bit subtract, low level

Module RETURN
Program 8
Data area 0
Entries: RM, RN
Zero and -1 standard returns

Module **G** Program 6, Data area 0 Entries: C.SXT, G.	Fetch character and sign extension
Module **INPORT** Program 14, Data area 0 Entries: INPORT	Input byte from CPU port
Module **OUTPOR** Program 17, Data area 0 Entries: OUTPOR	Output byte to CPU port
Module **CTST** Program 6, Data area 0 Entries: C.TST	Utility sign test, low level
Module **BINFLA** Program 6, Data area 2 Entries: GC_BIN, PC_BIN	Binary flags for I/O
Module **END** Program 6, Data area 0 Entries: $END	End routine, creates address for sbrk() to manage heap space.

Module **SSORT2** Shell sort for pointer tables
Program 215, Data area 16
Entries: SSORT2
Externals: $RM

Module **SSORT3** Shell sort for MBASIC structures
Program 237, Data area 0
Entries: SSORT3
Externals: $RM

Module **NUMSOR** Distribution sort (mathsort)
Program 184, Data area 10
Entries: NUMSOR

Module **TIMER** H89/Z100 timer utility 1/10 second
Program 209, Data area 2
Entries: TIMER
Externals: C.UDV, H., ITOA, PEEKW, PUTCHA, Q., S.

Module **DSORT1** Distribution sort, 16-bit numbers
Program 176, Data area 10
Entries: DSORT1
Externals: MOVE$

Module **DSORT** Distribution sort, single byte
Program 152, Data area 10
Entries: DSORT
Externals: MOVE$

Module **RAWIO** rdDISK, wrDISK direct-disk
Program 164, Data area 0
Entries: RDDISK, WRDISK

Module **WAITZ** H89/Z100 timeout wait routine
Program 25, Data area 0
Entries: WAITZ

Module **BINARY** Binary search for integers
Program 96, Data area 8
Entries: BINARY
Externals: $RM, C.LE, C.LT, H., S.1

Module **SBINAR** Binary search for strings
Program 99, Data area 8
Entries: SBINAR
Externals: $RM, C.LE, C.LT, H., STRCMP

Module **SEEKZ** Low-level integer seek,ftell,ftellr
Program 1001, Data area 24
Entries: FTELL, FTELLR, SEEK
Externals: .SWITC, C.DIV, C.GT, C.LT, C.MULT, C.NOT, C.ULT, CCBDOS, E., E.0, FFB, FFLUSH, G., H., IOBIN, IOEND, IOFCB, IOIND, IOMODE, IOSECT, MOVEME, Q., S.

Module SFATN1 Scanf module
Program 245, Data area 10
Entries: SF_ATN, SF_OP1
Externals: .SWITC, ATOF, C.MULT, H., HI.BL, I..L, L.ADD, L.MUL, LLONG., Q., SLONG., ST4.

Module SFATN2 Scanf module
Program 217, Data area 10
Entries: SF_ATN, SF_OP2
Externals: .SWITC, C.MULT, H., HI.BL, I..L, L.ADD, L.MUL, LLONG., Q., SLONG., ST4.

Module SFATN3 Scanf module
Program 125, Data area 6
Entries: SF_ATN, SF_OP3
Externals: .SWITC, ATOF, C.MULT, H., Q., SLONG.

Module SCANF Main scanf routine
Program 1695, Data area 40
Entries: F_SCAN, SCAN_F, STK_PO, S_SCAN
Externals: .SWITC, C.GT, C.MULT, C.NOT, C.TST, C.UGE, E., E.O, G., GETC, GETCHA, H., INDEX, ISDIGI, ISSPAC, ISXDIG, N., SF_ATN, TOLOWE, UNGETC

Module SFATN4 Scanf module
Program 97, Data area 6
Entries: SF_ATN, SF_OP4
Externals: .SWITC, C.MULT, H., Q.

Module RUN Run a CP/M command line.
Program 51, Data 0
Entries: RUN
Externals: .XFCB, CBUF, CHAIN, FNB$, INSERT, SOB$, STRCPY

Module CHAIN Chain from FCB to tpa.
Program 94, Data 0
Entries: CHAIN
Externals: MOVE$

Module INSERT Insert command tail into default
Program 58, Data 0 CP/M buffers.
Entries: INSERT
Externals: .XFCB, CVUPPE, FNB$, MOVE$, SOB$, STRLEN

Module GETDDI Get directory info strings.
Program 387, Data 8
Entries: GETDDI
Externals: C.GT, C.LE, C.LT, C.NOT, DECODF, E., E.O, FINDFI, FINDNE, H., LOGGED, N., STRCMP, STRCPY

Module GETDFR Get disk free space info.
Program 400, Data 20
Entries: GETDFR
Externals: A., C.ASL, C.DIV, C.GE, C.LE, C.MULT, C.NOT, C.UDV, CCBDOS, CCBIOS, E.O, H., Q., TOUPPE

Module PEEKL — Peek at address, long return
Program 8, Data area 0
Entries: PEEKL
Externals: LLONG.

Module POKEL — Poke long at address
Program 15, Data area 0
Entries: POKEL
Externals: LLONG., SLONG.

Module PFOP1 — Printf module
Program 5, Data area 0
Entries: CVLTOA, PF_OP1

Module PFOP2 — Printf module
Program 5, Data area 0
Entries: CVFTOA, PF_OP2

Module PF — Printf main routine, long & float.
Program 1202, Data area 42
Entries: PRNT 1, PRNT 2, PRNT 3, PRNT 4
Externals: .SWITC, A., C.GE, C.GT, C.MULT, C.NEG, C.NOT, C.UDV, C.UGE, C.ULT, C.USR, CVFTOA, CVLTOA, E.0, H., MIN, PUTC, PUTCHA, S., STRLEN

Module CVLTOA — Convert long to ASCII for Printf()
Program 554, Data area 22
Entries: CVLTOA
Externals: BL.HU, C.NOT, C.ULT, CL.LT, E.0, H., HI.BL, HU.BL, L.AND, L.ASR, L.DIV, L.MUL, L.NEG, L.SUB, LLONG., LONG.0, SLONG., SWAP4.

Module CVFTOA — Convert float to ASCII, Printf()
Program 195, Data area 12
Entries: CVFTOA
Externals: C.LT, FTOA, G., LLONG., SLONG., STRLEN

Module ATOL — ASCII to long conversion, base 10
Program 124, Data area 8
Entries: ATOL
Externals: C..L, I..L, ISDIGI, L.ADD, L.MUL, LLONG., SLONG., SOB, ST4.

Module LTOA — Long to ASCII, base 10. Fast version.
Program 337, Data area 16
Entries: LTOA
Externals: C.GT, C.TST, CL.GE, CL.GT, CL.LE, CL.LT, E.0, H., HI.BL, ITOA, L.NEG, L.SUB, LLONG., SLONG., SWAP4.

Module LABS — Long absolute value
Program 25, Data area 0
Entries: LABS
Externals: CL.GT, HI.BL, L.NEG

Module FABS — Float absolute value
Program 25, Data area 0
Entries: FABS
Externals: CF.GT, F.NEG, HI.BF

Module FTOA — Float to ASCII, 4 arguments
Program 470, Data area 2
Entries: FTOA, FTOA
Externals: DIV10., FADD., FADDA., FCOMP., FLNEG., INXHR., LLONG., MOVFR., MOVRF., MOVRM., MUL10., QINT., SIGN.

Module ATOF — ASCII to float, Unix-style
Program 199, Data area 0
Entries: ATOF
Externals: DIV10., FADD., FLNEG., FLOAT., MOVRF., MUL10., PUSHF., ZERO.

Module FSTACK — FLIBRARY from Software Toolworks
Program 73, Data area 0
Entries: C..F, C@@F, F..C, F..I, F..L, F..U, F@@C, F@@I, F@@L, F@@U, F_STAK, I..F, I@@F, L..F, L@@F, U..F, U@@F
Externals: BF.BL, BF.HC, BF.HI, BF.HU, BL.BF, C.1632, C.3216, G., HC.BF, HER.32, HI.BF, HU.BF, MOVRM., SLONG.

Module FLTLIB — FLIBRARY from Software Toolworks
Program 1112, Data area 15
Entries: BF.BL, BF.HC, BF.HI, BF.HU, BF@BL, BF@HC, BF@HI, BF@HU, BL.BF, BL@BF, CF.EQ, CF.GE, CF.GT, CF.LE, CF.LT, CF.NE, CF@EQ, CF@GE, CF@GT, CF@LE, CF@LT, CF@NE, DIV10., DUM_, ERRCOD, F.ADD, F.DIV, F.MUL, F.NEG, F.NOT, F.SUB, F@ADD, F@DIV, F@MUL, F@NEG, F@NOT, F@SUB, FACL_, FACL_1, FACL_2, FAC_, FAC_1, FADD., FADDA., FCOMP., FCOMP@, FDIV_A, FDIV_B, FDIV_C, FDIV_G, FLNEG., FLOAT., FLT.0, FLT@0, FLT_PK, FMLT_1, FMLT_2, HC.BF, HC@BF, HI.BF, HI@BF, HU.BF, HU@BF, INXHR., MOVFM., MOVFM@, MOVFR., MOVFR@, MOVMF., MOVMF@, MOVRF., MOVRF@, MUL10., PUSHF., QINT., SAVE_, SAVE_1, SIGN., ZERO.
Externals: C.NEG, C.SXT, EQ.4, G., HC.BL, HI.BL, HU.BL, L.ADD, L.NEG, LLONG., MOVRM., NEQ.4, SLONG.

Module LANDSH — FLIBRARY from Software Toolworks
Program 141, Data area 0
Entries: L.AND, L.ASL, L.ASR, L@AND, L@ASL, L@ASR, L_SHIF
Externals: G.

Module LCOMP — FLIBRARY from Software Toolworks
FLIBRARY from Software Toolworks
Program 84, Data area 0
Entries: CL.GE, CL.GT, CL.LE, CL.LT, CL@GE, CL@GT, CL@LE, CL@LT, L_COMP
Externals: G., LLONG., LONG.0, SLONG., SWAPS.

Module LMISC — FLIBRARY from Software Toolworks
Program 372, Data area 0
Entries: L.DIV, L.MOD, L.MUL, L.OR, L.XOR, L@DIV, L@MOD, L@MUL, L@OR, L@XOR, L_MDIV
Externals: G.

Module LADSUB FLIBRARY from Software Toolworks
Program 46, Data area 0
Entries: L.ADD, L.COM, L.NEG, L.SUB, L@ADD, L@COM, L@NEG, L@SUB, L_ADSB
Externals: G.

Module LSTACK FLIBRARY from Software Toolworks
Program 104, Data area 0
Entries: C..L, C.1632, C.3216, C@@L, HER.32, I..L I@@L, L..C, L..I,
L..U, L@@C, L@@I, L@@U, L_STAK, U..L, U@@L
Externals: BL.HC, BL.HI, BL.HU, G., HC.BL, HI.BL, HU.BL, MOVRM., SLONG.

Module FOURB FLIBRARY from Software Toolworks
Program 107, Data area 0
Entries: EQ.4, EQ@4, FOUR_B, L.NOT, L@NOT, LLONG., LLONG@, LONG.0,
LONG@0, MOVRM., MOVRM@, NEQ.4, NEQ@4, SLONG., SLONG@, ST4., ST4@,
SWAP4., SWAP4@, SWAPS., SWAPS@
Externals: G.

Module HTOBL FLIBRARY from Software Toolworks
Program 35, Data area 0
Entries: BL.HC, BL.HI, BL.HU, BL@HC, BL@HI, BL@HU, HC.BL, HC@BL, HI.BL,
HI@BL, HU.BL, HU@BL, I_LONG
Externals: C.SXT G.

CLIBU.REL -UNIX LIBRARY

Module UFLUSH Unix-style fflush
Program 54, Data area 0
Entries: UFLUSH
Externals: $RM, CCBDOS, FFLUSH, H., IOCH, IOFCB, MAXCHN, RN$

Module ABORT Abort debugger service
Program 6, Data area 0
Entries: ABORT

Module ACCESS Access system call
Program 105, Data area 4
Entries: ACCESS
Externals: C.NOT E.0 GETATT H.

Module CALLOC Memory allocation
Program 34, Data area 0
Entries: CALLOC
Externals: C.MULT, FILLCH, MALLOC

Module CHMOD Change file mode
Program 87, Data area 2
Entries: CHMOD
Externals: H., SETATT

Module CHOWN — Change file owner
Program 26, Data area 0
Entries: CHOWN
Externals: FINDFI, H.

Module CLOSE — Close file, low-level file descriptor
Program 3, Data area 0
Entries: CLOSE
Externals: FCLOSE

Module CREAT — Create file, low-level file descriptor
Program 122, Data area 2
Entries: CREATA, CREATB
Externals: C.TST, FOPEN, H., IOBIN, IOMODE

Module CTYPE — Character class functions, array type
Program 147, Data area 0
Entries: CTYPE_, ISTYPE

Module EXPAND — Command line expansion
Program 495, Data area 12
Entries: EXPAND
Externals: C.GT, CORE, CVUPPE, DECODF, E.O, FINDFI, FINDNE, H., INDEX, N., Q., STRCMP, STRCPY, STRLEN

Module ETEXT — Etext, End, Edata variables
Program 6, Data area 0
Entries: EDATA, END, ETEXT
Externals: $END, TOT_SZ

Module OPENB — Open low-level by file descriptor, binary mode.
Program 106, Data area 2
Entries: OPENB
Externals: C.GE, C.LE, E.O, FOPEN, H.

Module OPENA — Open low-level by file descriptor, ASCII
Program 103, Data area 2
Entries: OPENA
Externals: C.GE, C.LE, E.O, FOPEN, H.

Module FDOPEN — File descriptor open
Program 83, Data area 0
Entries: FDOPEN
Externals: C.GT, C.LE, DECODF, E.O, FREOPA, H., IOFCB, MAXCHN

Module FREOPB — Stream re-open binary mode
Program 179, Data area 6
Entries: FREOPB
Externals: E.O, FREOPA, H.

Module FREOPA — Stream re-open ASCII mode
Program 289, Data area 4
Entries: FREOPA
Externals: C.GT, C.LE, E., E.O, FCLOSE, FIN, FOPENA, FOUT, H., IOCH, MAXCHN

Module FEOF — End of file check
Program 90, Data area 2
Entries: FEOF
Externals: C.NOT, E.O, GETCBI, H., IOMODE, UNGETC

Module FERROR — File error reporting
Program 4, Data area 0
Entries: CLEARE, FERROR

Module FILENO — Stream to file descriptor conversion
Program 5, Data area 0
Entries: FILENO

Module FOPENB — Stream open binary mode
Program 170, Data area 6
Entries: FOPENB
Externals: E.O, FOPENA, H.

Module FOPENA — Stream open ASCII mode
Program 297, Data area 10
Entries: FOPENA
Externals: C.GT, C.LT, C.NOT, C.TST, E., E.O, FOPEN, H., IOMODE, MAXCHN, SEEKEN

Module SEEKEN — Seek to end of file, used for APPEND
Program 235, Data area 8
Entries: SEEKEN
Externals: C.UGT, CCBDOS, E.O, FFB, H., IOBIN, IOBUF, IOEND, IOFCB, IOIND, IOSECT, IOSIZE, Q.

Module FREAD — Unix-style fread()
Program 114, Data area 8
Entries: FREAD
Externals: C.GT, C.NOT, GETC H.

Module FSEEK — Unix-style fseek(), long integer seek()
Program 145, Data area 4
Entries: FSEEK
Externals: BL.HU, H., HI.BL, L.DIV, L.MOD, LLONG., Q., SEEK

Module FTELLU — Unix-style ftell(), file position
Program 132, Data area 8
Entries: FTELLU
Externals: E.O, FTELL, FTELLR, H., HI.BL, HU.BL, L.ADD, L.MUL, LLONG., SLONG., ST4.

Module FWRITE Unix-style fwrite()
Program 108, Data area 8
Entries: FWRITE
Externals: C.GT, H., PUTC

Module GETL Get a long integer from stream
Program 88, Data area 8
Entries: GETL
Externals: C.NOT, GETC, H., LLONG., ST4.

Module GETPAS Get password no-echo from user
Program 137, Data area 13
Entries: GETPAS
Externals: C.TST, CCBIOS, H., INDEX, PUTCHA, Q.

Module GETPID Get process ID, fake function
Program 4, Data area 0
Entries: GETPID

Module GETW Get word from stream
Program 78, Data area 6
Entries: GETW
Externals: C.NOT, C.TST, GETC, H.

Module HEAPS Heap sort
Program 484, Data area 14
Entries: HEAPS
Externals: C.DIV, C.GE, C.GT, C.TST, E.0, H., Q.

Module ISATTY Is a TTY system call
Program 43, Data area 0
Entries: ISATTY
Externals: C.NOT, E.0, H.

Module REALLO Re-allocate, Unix-style
Program 110, Data area 2
Entries: REALLO
Externals: E., FREE, H., MALLOC, MOVEME

Module MALLOC Memory allocate
Program 505, Data area 14
Entries: FREE, F RE L, MALLOC
Externals: C.UGE, C.UGT, C.ULE, C.ULT, E., E.0, H., Q., S., SBRK

Module MKTEMP Make a temp file name from template
Program 128, Data area 4
Entries: MKTEMP
Externals: C.TST, H., ITOA, STRLEN

Module PERROR Print system error message
Program 450, Data area 0
Entries: PERROR, SYS ER, SYS NE
Externals: C.GT, CON, E., ERRNO, H., Q.

Module PUTL — Put long integer out to stream
Program 75, Data area 4
Entries: PUTL
Externals: H., HI.BL, LLONG., PUTC

Module PUTW — Put 16 bit word out to stream
Program 84, Data area 4
Entries: PUTW
Externals: C.TST, H., PUTC

Module QSORT — Quick sort, Unix-style arguments
Program 906, Data area 15
Entries: QSORT
Externals: C.DIV, C.GE, C.GT, C.LE, C.LT, C.MULT, E.0, H., S.

Module QUICK — Quick sort, alternate arguments
Program 676, Data area 10
Entries: QUICK
Externals: C.DIV, C.GE, C.GT, C.LE, C.LT, E.0, H., Q., S.

Module READA — Read ASCII, Unix-style
Program 80, Data area 4
Entries: READA
Externals: C.GT, C.NOT, GETC, H.

Module READB — Read binary, modeled after READA
Program 80, Data area 4
Entries: READB
Externals: C.GT, C.NOT, GETCBI, H.

Module SETBUF — Set file buffer location
Program 71, Data area 0
Entries: SETBUF
Externals: C.UGE, C.ULE, E.0, H., IOBUF, MAXCHN, Q., REWIND

Module SHELLS — Shell sort
Program 343, Data area 12
Entries: SHELLS
Externals: C.DIV, C.GE, C.GT, C.TST, H., Q., S.

Module SIGNAL — Signal interrupt process, fake
Program 4, Data area 0
Entries: SIGNAL

Module SWAB — Swap bytes in buffer to buffer
Program 110, Data area 4
Entries: SWAB
Externals: C.GT, H., Q.

Module SYSTEM — System call for CP/M 2.2
Program 334, Data area 6
Entries: SYSTEM
Externals: C.GT, C.NOT, C.TST, CON, E., E.O, EXIT, FCLOSE, FOPEN, H., Q., REVERS, STRLEN, WRITE

Module TTYNAM — Get tty name
Program 82, Data area 0
Entries: TTYNAM
Externals: .SWITC, H.

Module WRITEA — Write function ASCII, Unix-style
Program 75, Data area 4
Entries: WRITEA
Externals: C.GT, H., PUTC

Module WRITEB — Write binary, modeled after WRITEA
Program 75, Data area 4
Entries: WRITEB
Externals: C.GT, H., PUTCBI

Module GETATT — Get file attributes for CP/M
Program 131, Data area 4
Entries: GETATT
Externals: E., FINDFI, H.

Module SETATT — Set file attributes for CP/M
Program 171, Data area 4
Entries: SETATT
Externals: A., C.TST, CCBDOS, H., MAKEFC

Module BSEARC
Program 178, Data area 8
Entries: BSEARC
Externals: C.ASR, C.MULT, C.NOT, C.TST, H., Q., S.

Module LSEARC
Program 160, Data area 4
Entries: LSEARC
Externals: C.NOT, C.TST, H., MOVEME, Q.

Module UNIXIO
Program 15, Data area 0
Entries: ACCT, ALARM, CUSERI, DUP, DUP2, ENDGRE, ENDPWE, FORK, FSTAT, FTIME, GETEGI, GETEUI, GETGID, GETGRE, GETGRG, GETGRN, GETLOG, GETPGR, GETPID, GETPPI, GETPW, GETPWE, GETPWN, GETPWU, GETUID, IOCTL, KILL, LINK, , MKNOD, MONITO, MOUNT, NICE, PAUSE, PCLOSE, PIPE, POPEN, PROFIL, PTRACE, PUTPWE, SETGID, SETGRE, SETPGR, SETPWE, SETUID, SLEEP, STAT, STIME, SYNC, TIMES, UMASK, UMOUNT, UNAME, USTAT, UTIME, WAIT

Module **RAND** Random number generator
Program 267, Data area 118
Entries: RAND, RAND1, SRAND
Externals: C.MULT, H., Q.

Module **ACOT** Arc cotangent
Program 29, Data area 0
Entries: ACOT
Externals: ATAN, F.ADD, F.NEG, LLONG.

Module **ACOS** Arc cosine
Program 168, Data area 4
Entries: ACOS
Externals: ATAN, CF.EQ, CF.GT, ERRNO, F.DIV, F.MUL, F.NEG, F.SUB, FABS, HI.BF, LLONG., SLONG., SQRT, SWAP4.

Module **ACSC** Arc cosecant
Program 162, Data area 4
Entries: ACSC
Externals: ASEC, CF.EQ, CF.LE, CF.LT, ERRNO F.DIV, F.MUL, F.NEG, F.SUB, LLONG., SLONG., SQRT, SWAP4.

Module **ASIN** Arc sine
Program 160, Data area 4
Entries: ASIN
Externals: ATAN, CF.EQ, CF.GT, ERRNO, F.DIV, F.MUL, F.NEG, F.SUB, HI.BF, LLONG., SLONG., SQRT, SWAP4.

Module **ASEC** Arc secant
Program 128, Data area 4
Entries: ASEC
Externals: ATAN, CF.LE, CF.LT, ERRNO F.MUL F.SUB LLONG. SLONG., SQRT, SWAP4.

Module **ATAN2** Arc tangent of y/x
Program 122, Data area 0
Entries: ATAN2
Externals: ATAN, CF.EQ, CF.LT, ERRNO, F.DIV, F.NEG, LLONG.

Module **ATAN** Arc tangent
Program 321, Data area 12
Entries: ATAN
Externals: CF.EQ, CF.GT, CF.LT, ERRNO F.DIV, F.MUL, F.NEG, F.SUB, HORNER, LLONG., SLONG.

Module **LDEXP** Load exponent
Program 31, Data area 0
Entries: LDEXP
Externals: H., LLONG.

Module FREXP — Fraction exponent
Program 52, Data area 0
Entries: FREXP
Externals: A., H., LLONG., Q., S.

Module TAN — Tangent
Program 458, Data area 10
Entries: TAN
Externals: ABS, BF.HI, C.TST, CF.EQ, CF.GT, COS, ERRNO, F.ADD, F.DIV, F.MUL, F.NEG, F.SUB, HORNER, I..F, LLONG., SIN, SLONG., SWAP4.

Module COS — Cosine
Program 26, Data area 0
Entries: COS
Externals: F.ADD, LLONG., SIN

Module SIN — Sine
Program 357, Data area 12
Entries: SIN
Externals: .SWITC, BF.HI, CF.LE F.DIV, F.MUL F.NEG F.SUB, HI.BF, HORNER, I..F, LLONG., SLONG., SWAP4.

Module SQRT — Square root
Program 392, Data area 14
Entries: SQRT
Externals: CF.EQ, CF.GT, CF.LT ERRNO, F.ADD, F.DIV, F.MUL, HI.BF, HORNER, LLONG., SLONG.

Module SINH — Hyperbolic sine
Program 122, Data area 0
Entries: SINH
Externals: CF.GT, CF.LT, ERRNO, EXP, F.DIV, F.NEG, F.SUB, LLONG.

Module COSH — Hyperbolic cosine
Program 89, Data area 0
Entries: COSH
Externals: CF.GT, ERRNO, EXP, F.ADD, F.DIV, F.NEG, FABS, LLONG.

Module TANH — Hyperbolic tangent
Program 159, Data area 4
Entries: TANH
Externals: CF.GT, CF.LT, ERRNO, EXP F.ADD F.DIV, F.NEG, F.SUB, LLONG., SLONG., SWAP4.

Module EXP — Natural base exponential
Program 97, Data area 0
Entries: EXP
Externals: CF.GT, CF.LT, ERRNO, F.MUL, F.NEG, LLONG., POW10

Module POW — Power function x to the y
Program 521, Data area 14
Entries: POW
Externals: BF.BL, BF.HI, BL.BF, C.GT, CF.EQ, CF.GT, CF.LT, E.0, ERRNO, F.DIV, F.MUL, F.NEG, FABS, G. HI.BF HI.BL L.AND, LLONG., LOG10, LONG.0, POW10, SLONG.

Module POW10 — Power function 10 to the y
Program 534, Data area 10
Entries: POW10
Externals: BF.HI, C.DIV, C.TST, CF.GT, CF.LE, CF.LT F.DIV F.MUL, F.NEG, F.SUB, FLT.0, HI.BF, HORNER, I..F, LLONG., SLONG.

Module LOG — Natural logarithm base e
Program 26, Data area 0
Entries: LOG
Externals: F.MUL, LLONG., LOG10

Module LOG10 — Standard logarithm base 10
Program 370, Data area 22
Entries: LOG10
Externals: A., CF.EQ, CF.GE, ERRNO, F.ADD, F.DIV, F.MUL, F.NEG, F.SUB, HORNER, I..F, LLONG., S., SLONG., SWAP4.

Module CEIL — Long integer ceiling
Program 77, Data area 4
Entries: CEIL
Externals: BF.BL, BL.BF, CF.LT, HI.BL, L.ADD, LLONG., SLONG.

Module FLOOR — Long integer floor
Program 125, Data area 4
Entries: FLOOR
Externals: BF.BL, BL.BF, CF.GT, CL.LT, HI.BL, L.ADD, L.SUB, LLONG., SLONG.

Module FMOD — Float remainder function
Program 99, Data area 4
Entries: FMOD
Externals: BF.BL, BL.BF, CF.EQ, ERRNO, F.DIV, F.MUL, F.SUB, LLONG., SLONG., SWAP4.

Module MODF — Remainder function for integer part
Program 103, Data area 6
Entries: MODF
Externals BF.HI, CF.LT, ERRNO, F.SUB, FABS, H., HI.BF, I..F, LLONG., Q., SLONG., SWAP4.

Module RAD — Degrees to radians
Program 19, Data area 0
Entries: RAD
Externals: F.MUL, LLONG.

Module DEG — Radians to degrees
Program 19, Data area 0
Entries: DEG
Externals: F.MUL, LLONG.

Module LABS — Long absolute value
Program 42, Data area 0
Entries: LABS
Externals: CL.GE, HI.BL, L.NEG, LLONG.

Module FABS — Float absolute value
Program 42, Data area 0
Entries: FABS
Externals: CF.GE, F.NEG, HI.BF, LLONG.

Module ABS — Integer absolute value
Program 17, Data area 0
Entries: ABS
Externals: C.LE, C.NEG

Module HORNER — Horners method for polynomial evaluation
Program 104, Data area 4
Entries: HORNER
Externals: F.ADD, F.MUL, H., LLONG., Q., SLONG.

INDEX TO THE USER GUIDE

#asm,#define,#endasm,#endif ... 8
#if,#ifdef,#ifndef,#include,#undef ... 5,15
$$$.SUB ... 5
$B8## ... 39
$LM ... 27,42
$OFF&X ... 42
$OFF1,$OFF2,$OFF3,$OFF4,$OFF5,$OFF6 ... 26,27
$PCH## ... 39
SIZ,$SIZ&X,$SIZ1,$SIZ2,$SIZ3,$SIZ4,$SIZ5,$SIZ6 ... 26,27,42
$TAG&X,$SIZ&X,$OFF&X ... 42
$TAG1,$TAG2,$TAG3,$TAG4,$TAG5,$TAG6 ... 26,27
%f,%ld ... 15
'\n' ... 20
(FLOAT)1.570796326795,(FLOAT)3.14159265359 ... 21
-Abackup.arc ... 24
.PRINTX (M80)... 40,42
.REQUEST (L80)... 16,40
2-drive ... 1
2.203 ... 3
3.14159 ... 8
3.44 ... 2
3.45 ... 2,4
320k ... 3
32767 ... 8
4096 ... 7,10,11,14
4MHZ ... 7
<GRP.H> ... 36
<lib1>,<lib2>,<lib3> ... 42
<PWD.H> ... 36,37
<SETJMP.H> ... 20
<STDIO.H> ... 4,5,6,7,9,10,11,12,13,14,15,20,31,33
<SYS/STAT.H> ... 34,35
<SYS/TIMEB.H> ... 34,35
<SYS/TYPES.H> ... 34,35,36
<UNIX.H> ... 18,33
<VG.H> ... 13
a:$$$.sub ... 5
alarm(seconds) ... 33
arc.com ... 1,24,25
archive ... 1,23,24,25
assembler ... 1,3,29
atof ... 17,41
atoi() ... 17
atol ... 17,41
auto-request ... 9
automatically ... 9,12,13,15,16,24,27
a_exit ... 27,39
back-up ... 24
background ... 30
Banahan ... 33
BDOS ... 6,10,42
Bourne's ... 6

buffers ... 7,9,11,14,15,26,27,31
bufsz1 ... 7,11,14,42
BUGS ... 2
c.com ... 1,2,3
C/80 ... 1,2,3,4,8,11,12,16,17,18,19,27,30,31,41,42
calloc() ... 18
cbuf ... 27,31
cc.com ... 1,2,3
CCMAIN.REL ... 27,42
cconfig.com ... 1,2,3
ccp ... 10,14,32
ccspace ... 10,11,14,31,32,39,42
Cexit () ... 10,40
cfree() ... 18
chain ... 10,14,17,41
character-only ... 39
chario ... 6,14,39
CL.COM ... 1
CLIB.REL ... 1,2,22,26,27,28,39
CLIB/L ... 28
CLIB/U ... 27
CLIBM ... 1,9,12,13,15,17,23,27,39
CLIBM0.REL,CLIBM1.REL ... 1
CLIBT ... 1,9,17,21,22,23,28
CLIBT.ARC ... 28
CLIBU ... 1,9,17,18,22,33
CLIBX ... 1,9,17,27
CLINK ... 1,4,5,12,13
Cmode ... 27
cmp.c ... 2
compilation ... 4
con() ... 6
configuration ... 1,3,31
conout(),cono 1(),cono 2 ... 16,39
console ... 6,7,16,24,32,39
conversion ... 3,6,18
Copen ... 40
copier ... 1,3,7
cp ... 1,2,3,5,10,14,24,31,32,33,41
CP/M-86 ... 5
CPU-dependent ... 8
CtlB ... 27
Ctrl-a,Ctrl-c,Ctrl-j,Ctrl-z ... 20,24
CTYPE.H ... 1
cycle ... 12
C mcln ... 40
DDT.COM ... 2
debugging ... 6,12
December-1981 ... 2
DecSystem ... 16
devices ... 7,8,19,26
diary ... 13

direct-disk ... 41
double ... 28
DRI ... 1
driver ... 16
dsort,dsort16 ... 17,41
dual-register ... 29
dup(old),dup2(old,new) ... 34
E2BIG,EACCES,EAGAIN,EBADF,EBUSY,ECHILD ... 22
echo.c ... 7
EDOM,EEXIST,EFAULT,EFBIG,EINTR,EINVAL ... 22
EIO,EISDIR,EMFILE,EMLINK ... 22
end-of-program ... 27
END.REL ... 27
endgrent() ... 36
endpwent() ... 37
ENFILE,ENODEV,ENOENT,ENOEXEC,ENOMEM ... 22
ENOSPC,ENOTBLK,ENOTDIR,ENOTTY ... 22
envl[1],env2[1] ... 20
ENXIO ... 22
EOF ... 7,16,39
EPERM,EPIPE ... 22
ERANGE ... 21,22
Eratosthenes ... 14
EROFS ... 22
errno.h ... 1,22
error(),err 1(),err 2 ... 4,39
ESPIPE,ESRCH,ETXTBSY ... 22
EX.COM ... 1
EXDEV ... 22
exit() ... 11,20
exitto(addr) ... 10,30
exp ... 23
EXPLARGE ... 21
extern ... 10,11,12,13,22,26,31
EX_SPC,TOT_SZ ... 27,42
f(T ... 13
FALSE ... 14,39
fdexp() ... 28
FDOS ... 9,10,31,32
fgetc() ... 18,19
filecopy ... 7
fin,fin_() ... 14,27
fixed-origin ... 27
flag ... 2,7,17,40
FLIBRARY.REL ... 1
float ... 2,9,15,17,21,23,28,29,39,40,41
floating-point ... 15
flush ... 14
fopen(),fopena(),fopenb() ... 18
force-searched ... 27
fork() ... 34
FORLIB.REL ... 27

formats ... 3,15
FORTRAN ... 16,27
fout,fout_() ... 14,27
fprintf ... 9,31,39
fprt_1(),fprt_2 ... 39
fputs() ... 6
free ... 4,9,14,18,26
freopa,freopb(),freopen ... 18
frexp() ... 28
fscanf ... 9,15,39
fseek ... 17,18,41
fstat(fd,buf) ... 34
ftell(),ftellu() ... 17,18,41
ftime(tp) ... 34
ftoa ... 17,41
g() ... 13
getc() ... 18,19
getcbinary(fp) ... 20
getegid() ... 34
geteuid() ... 34
getgid() ... 34
getl ... 17,41
getpgrp() ... 34
getpid() ... 34
getppid() ... 34
getpw(uid,buf) ... 36
gets() ... 6,16
getuid() ... 34
global ... 12,25,28,29,33
go.com ... 1,5
graphics ... 6,16
h() ... 13
h.## ... 30
H37 ... 3
H89,Z90 ... 1,3,7,41
handle ... 2,24,39
heap ... 15,26,27,31,32
HELLO.C ... 4,5,6,7
hooks ... 9,17,18
IF1,IF2,IFDEF,IFNDEF ... 40,42
incremental ... 12,13
indirection ... 32
insert ... 13,17,18,40,41,42
INTEL ... 25,28
IObuf[fd] ... 26,32
IOch[] ... 26
ioctl(fd,request,argp) ... 34
IOfcb[fd] ... 26,32
IOfcb[],IObuf[] ... 26,31
IOnchx,IOsize ... 26
IOTABLE ... 2,15,26
IPIby180 ... 21

IRP ... 40,42
itoa() ... 17
jmpbuffer ... 20
jmp_buf ... 20
K&R,Kernighan ... 4,5,17
kill(pid,sig) ... 34
L80 ... 1,2,5,12,17,25,27
landmarks ... 2
LIB.COM ... 1,26,27
lib1,lib2,lib3 ... 9,12,13,16,27,42
LIBC.ARC,LIBCC.ARC ... 25,26
librarian ... 1
libraries ... 9,12,17,25,27
library-dependent ... 8
link(path1,path2) ... 34
LINK-80 ... 4
linker ... 1,2,12
LLONG ... 17,41
ln ... 21
loaders ... 10
log ... 21,23
long/float ... 2
longjmp(env1,10) ... 20
lseek() ... 18
ltoa((long)x,buffer) ... 13,17,41
M80 ... 1,2,14,17,40,42
macro ... 3,8,16,17,18,19,26,40,42
Macro-80 ... 3
MAKEWORK.SUB ... 1,3
malloc() ... 18
manager ... 1,26
MARC.COM ... 1,24
MATH.H,MATHDEF.H ... 1,17,21
mathlib ... 15,42
MathPack ... 2,17,23
MAX2EXP ... 21
MAXCHN,MAXCHN-1 ... 15,26
memmap.c ... 31,32
Microsoft ... 1,3,4,25,26,27,40
mknod(path,mode,dev) ... 34
MLIB ... 17,42
modules ... 12,13,14,15,25,27
monitor(lowpc,highpc,buffer,bufsize,nfunc) ... 33,37
mount(spec,dir,rwflag) ... 34
multi-argument ... 9,39
multiply-defined ... 33
newlines ... 16
nfiles ... 7,11,15,26,39,42
nice(incr) ... 35
non-float ... 9
NULL ... 16,39,40
nullcmd ... 5,6,11,14,40

numsort ... 17,41
obsolete ... 18
open(),opena(),openb() ... 15,18,19,26
optimize ... 40
overlay ... 10,11
passwd ... 35,36,37
pause() ... 35
PC ... 2
pclose(fp) ... 35
peekl ... 17,41
peekw(1) ... 10,11
peekw(6) ... 31
performance ... 7,15
perror() ... 22
pfFLOAT,pfLONG ... 9,15,39,40
PF_OP1,PF_OP2 ... 40
PIby180,PIbyTWO ... 21
PIP ... 1,3,24
pipe(fd) ... 35
pokel ... 17,41
port ... 18
portability ... 23
POW10INF ... 21
pre-processor ... 8,9,11,14,16
printf ... 4,5,6,9,13,15,16,28,39,40,41
printf,prnt_1(),prnt_2,prnt_3,prnt_4 ... 39
profil(buff,bufsize,offset,scale) ... 35
profile ... 2
printf,prtf_1(),prtf_2 ... 39
ptrace(request,pid,addr,data) ... 35
putc(c,stdout) ... 8
putchar() ... 6,7,8,16
putl ... 17,41
putpwent(p,fp) ... 35
puts ... 6,10,11,16,20,31,39
puts(p-4096+1+strlen("keyword")) ... 11
p_fin,p_fout ... 31
quad ... 7
randl,random ... 17,41
rdDISK ... 17,41
re-direction ... 7,9,10,11,14,40
re-open ... 18
read,reada,readb ... 11,14,18,19,21,41
REL ... 2,12,25
response ... 7
rewind() ... 26
Ritchie ... 4
RN$## ... 40
ROM,ROM-based,romable ... 3,16,25
run("overlay") ... 10
Rutter ... 33
sample=clib<sample>/e ... 28

sbinary ... 17,41
scanf() ... 9,15,39,40,41
searches ... 12,13,17
seek ... 17,41
Sep-1982 ... 2
setbuf() ... 26
setgid(gid) ... 35
setgrent() ... 36
setjmp(),setjmp.h,longjmp ... 1,20
SETLIB ... 17,40,41
setpgrp() ... 35
setpwent() ... 37
setuid(uid) ... 35
sfFLOAT,sfLONG ... 9,15,40
sfree() ... 26,27
SF_OP1,SF_OP2,SF_OP3 ... 40
sgtty ... 34
sieve ... 14
signal ... 1
SIGNAL.H ... 1
sleep(seconds) ... 37
SLONG ... 17,41
soft-sectored ... 7
SPHL ... 10,30
sprintf,sprt_1(),sprt_2 ... 9,39
sscanf ... 9,15,39
ssort2,ssort3 ... 17,41
stack ... 10,14,29,30,31,32
stat(name,buf) ... 35
stderr,stdin,stdout ... 10,14,16,39
stime(tp) ... 35
STK_POS,f_scan,scan_f,s_scan ... 17,39
stream ... 7,16,19,26,39,41
style ... 11
SUBMIT.COM ... 1
subroutines ... 9,14
switch(setjmp(env1)),switch(setjmp(env2)) ... 20
switches ... 9,11,12,14,15,16,17,26,39,40
symbol ... 3,8,9,17,21,25,27,40
SYMBOLS.ARC ... 25
SYMBOLS.SYM ... 25
sync() ... 35
sysgen,sysgened ... 1,3
test1.c,test2.c ... 7
time(tloc) ... 35
timer ... 17,41
times(buffer) ... 35
TLIB ... 17,21,41,42
tokens ... 40
tolower ... 18
Toolworks ... 1,2,4,17,19,23
TOT_SZ ... 27,42

toupper ... 18
TRACE ... 2
transcendental ... 1,23,42
TREE.C ... 2
trial,trial/n/y/e ... 25
trial.c,trial.sym ... 25
trick ... 5
TRUE ... 1,6,8,14,39
TWObyPI ... 21
type ... 4,12,13,16,28,39
type,type_1(),type_2,type_3 ... 39
typedef ... 8
uid ... 35,36,37
ULIB ... 17,42
umask(cmask) ... 36
umount(spec) ... 36
uname(name) ... 36
unbuff(fd) ... 26
Unix ... 1,6,14,15,16,17,18,19,22,23,33,39,41,42
Unix-compatible ... 39
Unix-standard ... 15
UNIX.H ... 1,18,19
UNIXIO.H,UNIXIO.REL ... 33
uppercase ... 14
USELIB ... 17,40,42
ustat(dev,buf) ... 36
utime(path,times) ... 36
V,V1,V2,V3,CLIBM/S,VG,CLIB/S,V/n/e ... 12
V.C,V.COM,V1,V2,V3,VG.C,VG.H,VG.REL ... 12,13,42
void ... 34,35,39
wait(2),wait(x) ... 33
wait(stat_loc) ... 36
waitz,waitz(500*x) ... 17,33,41
wildcards ... 24
Wiley ... 33
winchester ... 7
wrDISK ... 17,41
write,writea,writeb ... 5,14,18,19,22,24,41
x,<1,2,3,4,5,6> ... 42
XDIR.COM,XDIRUTIL.COM ... 1
XLIB ... 17,42
XREFSYM.C,XREFSYM.COM ... 25
Z90/Z100 ... 41
_exit, _exit() ... 14,39
_tolower, _toupper ... 18

CLIBRARY VERSION 3.2

Library by: G.B.Gustafson
2243 South Belaire Drive
Salt Lake City, UT 84109
1-801-466-6820

Compiler: C/80, Version 2.0/3.0/3.1,
by The Software Toolworks

Assembler: M80.COM, by Microsoft

Linker: L80.COM, by Microsoft

Librarian: LIB.COM, by Microsoft

Copyright 1985 by G.B.Gustafson and Viking C Systems. All rights reserved.

Trademarks. CP/M, CP/M-68K, DDT, SID are trademarks of Digital Research, Incorporated. Unix, Unix V7, Unix V5, Unix V3, PCC are trademarks of Bell Laboratories, Incorporated. C/80, MathPack, Flibrary, PIE are trademarks of The Software Toolworks. MBASIC, FORLIB, M80, L80, LIB, F80, BASCOM, BASLIB, MS-DOS are trademarks of Microsoft, Incorporated.

MAKING A WORK DISK. Format and sysgen a work disk for drive A:, and copy these files onto the new disk, in the order given. Files marked with asterisk (*) are optional. Files marked with a pound sign (#) are utilities with a specific function - replace the name given with the one you normally use.

XDIR.COM	*#	Sorted directory program (source XDIRUTIL.COM)
GO.COM		Made with SAVE 0 GO.COM, 0 records.
CC.COM		Software Toolworks Compiler (See notes)
M80.COM		Microsoft assembler (See notes)
L80.COM		Microsoft linker (See notes)
PI.COM	*#	A text editor for source files
PIP.COM	*	DRI CP/M 2.2 copier for moving files
CP.COM	*	Copier for moving files
CLINK.COM		C compile & assemble & link
EX.COM	*	Memory submit program.
CL.COM	*	C compile & assemble & link, H89/Z90.
STDIO.H		Standard I/O header file
UNIX.H	*	Unix header file for CLIBU.REL
ERRNO.H	*	Error reporting header file for CLIBT.REL
MATH.H	*	Math package header file for CLIBT.REL
CTYPE.H	*	Array-based character class header file
SETJMP.H	*	Setjmp/longjmp header file
SIGNAL.H	*	Signal header file
CLIB.REL		Main library
CLIBX.REL		Extensions of the main library
CLIBM.REL	*	Math library, floats and longs (see notes)
CLIBU.REL	*	Unix library
CLIBT.REL	*	Transcendental function library
LIB.COM	*	Microsoft librarian
STAT.COM	*	DRI CP/M 2.2 STAT utility
ARC.COM	*	Archive manager, part I
MARC.COM	*	Archive manager, part II
CCONFIG.COM	*	C/80 Configuration for CC.COM

If you have a 2-drive system, then MAKEWORK.SUB and SUBMIT.COM may be used to do the work. This method is recommended if you are familiar with SUBMIT. If not, then ignore these remarks and use PIP to copy the files.

Notes:

- To make CC.COM from C.COM on the C/80 3.1 distribution disk, run the program CCONFIG.COM and set the M80 switch to TRUE. Save the file as CC.COM.

- **MAKING CLIBM.REL FROM CLIBM0.REL AND FLIBRARY.REL**

```
A>PIP CLIBM0.REL=B:CLIBM0.REL
A>PIP CLIBM1.REL=B:FLIBRARY.REL
A>LIB CLIBM=CLIBM0,CLIBM1/E
A>ERA CLIBM0.REL
A>ERA CLIBM1.REL
```

- The best version of M80 is December-1981, 3.44. Others cannot handle lowercase, or have bugs when used with submit utilities.

- The best version of L80 is Sep-1982, 3.45. Earlier versions cannot link some of the REL files in CLIB.REL. The problem is in external arithmetic. The only cure is a good linker. To understand the problem, look at IOTABLE.CC in the sources.

- Some version 3.0 distribution disks had a file CCONFIG.COM that omitted long and float code generation options. The correct CCONFIG for floats and longs appeared on the MATHPACK disk.

- Both version 2.0 and 3.1 compilers generate source code that is compatible with this library. Some of the C/80 sources are not duplicated in this package, notably the PROFILE and TRACE sources and sample programs like CMP.C and TREE.C.

- Compilers have bugs. If you find one that doesn't, let everyone know. We'll buy it! If your C/80 compiler is more than a few months old, send $12.00 to The Software Toolworks for an update. The growth of the long/float compiler had its landmarks - get the latest version.

- **MAKING CC.COM FROM C.COM under C/80 version 2.0**
  The file CC.COM is made from C.COM on the C/80 distribution disk by patching the M80-compatibility flag, using the CP/M program DDT.COM. Here are the steps for **VERSION 2.0 ONLY:**

```
A>DDT C.COM
  DDT VERS 2.2
  NEXT  PC
  7C00 0100
  -s0143
  0143 00 1
  0144 00 .
  -g0
 A>save 123 CC.COM
```

o **MAKING CC.COM FROM C.COM under C/80 version 3.0+**
The file cc.com is a copy of c.com on the distribution disk, made using PIP. Following the copy, run CCONFIG.COM, and set the variables as below. This configuration works for CP/M 2.203 on an H89 with 64k memory, H37 drives. Other configurations will require reductions in the sizes for A, B, D.

Programs with large string use may need to set A, D near 0 and increase B to 3000 or more.

C/80 CONFIGURATION:

A: Symbol table size (-s): 370
B: String constant table size (-c): 1000
C: Dump constants after each routine (-k): YES
D: Macro (#define) table size (-d): 2100
E: Switch table size (-w): 200
F: Structure table size (-r): 400
G: Merge duplicate string constants (-f): NO
H: Assembler (-m): **Microsoft Macro-80**
I: Initialize arrays to zero (-z): < 256 BYTES ONLY
J: Generate ROMable code in Macro-80 (CSEG/DSEG) (-a): YES
K: Screen size (for error msg pause; 0 for none) (-e): 0
L: Generate slightly larger, faster code (-o): YES
M: Sign extension on char to int conversion (-x): NO
N: Device for library files (-v): A:

o **Contents of MAKEWORK.SUB.** The submit file used to make a work disk assumes that drive A: contains a sysgened and formatted disk of size 320k or larger. Drive B: is to contain all the source files listed below. The service of the operation is to place the files on the work disk in optimal order. The file copier cp is supplied with the package.

```
B:CP A:=B:XDIR.COM,GO.COM,CC.COM,M80.COM,L80.COM
B:CP A:=B:PI.COM,PIP.COM,CP.COM,CLINK.COM,EX.COM,CL.COM
B:CP A:=B:STDIO.H,UNIX.H,ERRNO.H,MATH.H,CTYPE.H,SETJMP.H,SIGNAL.H
B:CP A:=B:CLIB.REL,CLIBX.REL,CLIBM.REL,CLIBU.REL,CLIBT.REL
B:CP A:=B:LIB.COM,STAT.COM,ARC.COM,MARC.COM,CCONFIG.COM
```

Kernighan & Ritchie HELLO WORLD. To create, compile, assemble and link the classic Kernighan and Ritchie program HELLO.C, start your text editor and type these lines:

```
#include <stdio.h>
main()
{
  printf("Hello World!\n");
}
```

COMPILE, ASSEMBLE, LINK. The process is handled completely by CLINK.COM, as follows:

```
A>CLINK HELLO
```

The result of CLINK is a SUBMIT run that prints the following:

```
A>clink hello

A>ERA HELLO.COM
NO FILE
A>CC HELLO=HELLO

C/80 Compiler 3.1 (3/10/84) - (c) 1984 The Software Toolworks

0 errors in compilation
1K bytes free

A>M80 =HELLO
* STDIO.H - Ver 3.2 *
* CLIB Request *

No Fatal error(s)

A>ERA HELLO.MAC
A>L80 HELLO,HELLO/N/E

Link-80  3.45  26-Sep-82  Copyright (c) 1981 Microsoft

Data    0103    0AB6    < 2483>

40072 Bytes Free
[0133   0AB6      10]

A>ERA HELLO.REL
```

The program HELLO.COM should run as follows:

```
A>HELLO
Hello World!

A>
```

OPTIONAL RUN METHOD. An optional method for running your program exists just after L80 finishes operation. This method requires that you have on disk a file called **GO.COM** created as follows:

```
A>SAVE 0 GO.COM
```

To start HELLO.COM just after L80 finishes the link, enter **go** as below to produce:

```
A>go
Hello World!

A>
```

This trick works with unbanked CP/M 2.2 only. Don't try it under CP/M-86 or CP/M 3.0. It can be repeated as many times as desired. The file GO.COM causes a jump to address 100 hex, which runs the currently loaded program.

SAVING CLINK COMMANDS. CLINK creates a file A:$$$.SUB with the desired commands. The lines of this command file are printed during execution. The contents of $$$.SUB may be saved in HELLO.SUB with this command line:

```
A>CLINK HELLO -S
```

CONTROL OF OBJECT CODE SIZE. The main objection of assembly language programmers to language C is the unreasonable overhead attributed to the C library. Some implementations cause the minimum executable file to be 16k bytes or larger.

In contrast, this library lets you link in just what you need to make the program run. As an example, here is how to write the K&R HELLO.C source in order to make it compile into an executable file of less than 256 bytes.

```
#define nullcmd 1
#include <stdio.h>
#define putchar con
#undef printf
printf(p) char *p; { while(*p) putchar(*p++); }
main()
{
  printf("Hello world\r\n");
}
```

AN EVEN SMALLER HELLO.COM. The above is not the smallest HELLO.COM possible. It is possible to reduce the size even further, as follows:

```
#define nullcmd 1
#include <stdio.h>
#define putchar con
main()
{
static char *p;
  p = "Hello world\r\n";
  while(*p) putchar(*p++);
}
```

The library function con() is a raw-mode function that prints characters to the console via a standard BDOS call. It is useful for debugging, graphics and conversion of assembly language code segments to C source files. It is the same function that an assembly language programmer would use to print to the console.

The same idea can be used with the library printf(), because it calls putchar(). Here is a re-coding of HELLO.C that compiles to less than 1024 bytes, yet still includes the entire printf support code.

```
#define nullcmd 1
#include <stdio.h>
putchar(x) int x; { con(x);}
main()
{
  printf("Hello world!\r\n");
}
```

USING PUTS. The well-known Unix functions gets() (get a string from the console) and puts() (put a string to the console) are included, true to the description in Bourne's book **The Unix System**. In particular, gets() eats the cr/lf on input and puts() appends a cr/lf on output. Here is HELLO.C coded with puts():

```
#define nullcmd 1
#define chario 1
#include <stdio.h>
main() { puts("Hello world!");}
```

USING FPUTS. The overhead of printf() can be avoided entirely by coding the C program with fputs(), provided that no numerical conversions are required. One such program is HELLO.C:

```
#define nullcmd 1
#define chario 1
#include <stdio.h>
main() { fputs("Hello world!\n",stderr);}
```

ECHO EXAMPLE. As a final example, consider the standard program echo. It is a disguised version of filecopy, due to the possibility of I/O re- direction.

```
#define C80 1
#if C80
#define nfiles 2          /* 2 pre-defined file buffers */
#define bufsz1 2048       /* stream 1 buffer is 2k      */
#define bufsz2 2048       /* stream 2 buffer is 2k      */
#endif
#include <stdio.h>
main()
{
static int x;
     while( (x = getchar()) != EOF) putchar(x);
}
```

If run with a standard command line, then each line typed at the console will appear twice, making for a not-so-interesting program. However, use of the re-direction flags > and < turns ECHO.C into a file copier.

To illustrate,

```
A>echo <test1.c >test2.c
```

copies input file **test1.c** to output file **test2.c**. Experiment with the idea, using devices lst: and con:, to obtain a print program and a console listing program from echo.c.

IMPROVING DISK PERFORMANCE. Settable buffer size makes for dramatic improvement in the response time of programs like ECHO. Experiment with ECHO.C using different buffer sizes. Optimization depends on the features of your disk controller and disk drives.

Drives rated at 96 tpi respond well with buffer size 4096. Ram disk does just as well with buffer size 128. Winchester disk does best with buffer size 1024, due to its internal buffering.

Compiler operation on an H89/Z90 machine at 4mhz is optimal with Quad soft-sectored disks using 1024 sectors (790k per disk). The best performance occurs with files copied in the order outlined above. In this case, HELLO.C will compile in 54 seconds. Other disk configurations and file orders can increase the compile time for HELLO.C by as much as 25 seconds (1 minute and 20 seconds total).

PRE-PROCESSOR DIRECTIVES

The pre-processor used by C/80 does not support code macros, e.g.,

```
#define putchar(c) putc(c,stdout)
```

will not compile under C/80.

C/80 does not support typedef, but the pre-processor may be used to get around typedef problems.

The most commonly used pre-processor directives are:

```
#define
#ifdef
#ifndef
#if
#endif
#asm
#endasm
#include
```

Pre-Processor Application Notes:

- The more pre-processor directives used, the less portable will be your code across all compilers.
- Across full C compilers, the reverse is true. For this sub-class of compilers, the pre-processor directives help to overcome problems with different CPU devices, addressing schemes and library interfaces.
- Use a #define directive whenever it is expected that the code being written is CPU-dependent, library-dependent or it uses information about the host system that could fail for another compiler.
- Commonly used constants like 3.14159 or 32767 should be supplied with a symbol, via #define.
- Array sizes and constant loop bounds should be given as a #define, in order to adjust usage uniformly across the source code.
- A #define need not contain constants. You may put variable names in the definition, even if they have not yet been defined in the source. The real restriction is usage: don't try to use something that has not been defined.

The header file STDIO.H contains many pre-processor directives and assembly language hooks. Its main job is:

1. Compile a module as MAIN or as a subroutine.
2. Auto-request libraries CLIBX, CLIBT, CLIBU, CLIBM.
3. Request user libraries through the lib1, lib2, lib3 pre-processor directives.
4. Switch long and float libraries on and off.
5. Set up multiple-argument functions like printf().
6. Turn on/off command line arguments and re-direction.
7. Set the amount of free space under FDOS.
8. Set the number of file buffers and their size.

The Version 3.2 library automatically determines the main program by checking for the appearance of the reserved symbol **MAIN**. If main() is not in the MAC file made by C80, then the file is compiled as a collection of subroutines.

By default, usage of printf and scanf select the non-float and non-long versions, in order to reduce the object code size.

To use float and/or long versions of printf and scanf, you must include switches in your source code, as follows:

```
#define pfFLOAT 1       /* Use floats in printf() */
#define pfLONG  1       /* Use longs in printf()  */
#define sfFLOAT 1       /* Use floats in scanf()  */
#define sfLONG  1       /* Use longs in scanf()   */
```

These switches must be known to the compiler by the time it reads STDIO.H. The same switches control code size for the functions fprintf(), sprintf(), fscanf(), sscanf().

The example below enables floats for printf() and longs for scanf(). There is a code size savings realized by not linking in irrelevant support code (e.g., we don't need longs in printf()).

```
#define pfFLOAT 1
#define sfLONG  1
#include <stdio.h>
main()
{
float f;
long L;
  while((scanf("%ld",&L)) > 0) {
    f = L;
    printf("f = %6.4f\n",f);
  }
}
```

Headroom under FDOS can be reserved, for the purpose of building loaders or to allow chained variables. In particular, you can leave the CP/M CCP resident, and exit to the CCP instead of to WARM BOOT.

Below, we illustrate how to avoid the exit to warm boot, and how to chain variables to an overlay. The method used avoids problems with re-direction: the **stdin** and **stdout** streams are closed on exit.

```
#define ccspace 2048        /* standard CP/M CCP size */
#include <stdio.h>
main()
{
extern char *CC_CCP;
     puts("Leaving MAIN, exit to CCP");
     exitto(CC_CCP);
}

exitto(address)
char *address;
{
     Cexit (0);      /* normal C stdin/stdout closes    */
     address;        /* load address of caller's stack  */
#asm
     SPHL            /* load stack pointer, pop and jump */
#endasm
}
```

The second use of **ccspace** is to communicate variables to an overlay. The idea is to set up a common block in the area between the C stack and BDOS. The overlay should check a keyword area to insure that the linkage is valid. Here's an example:

```
#define ccspace 4096     /* lots of room */
#include <stdio.h>
main()                   /* program EXAMPLE.COM links to OVERLAY.COM */
{
char *p,*peekw();
     p = peekw(1);       /* address of BDOS */
     strcpy(p-4096,"KeyWord");
     strcat(p-4096,"This the data in the common block");
     puts("Main running, running overlay.com");
     run("overlay");
}
```

Incremental development is recommended for programs that exceed 300 lines or 20k in source. While the C/80 compiler can compile large programs directly, the compile time will not be acceptable for debugging sessions. Use of libraries and subroutine modules can decrease the compile, assemble, link cycle time to a few minutes.

The scheme used for incremental development is to place all globals in one file, then invent a header file for use in other modules. The header file contains type declarations and extern definitions.

Modules that use the globals will #include the header file for the global variables. Even main() itself will #include the global header file. In this way, related sets of routines can be collected to a single file, compiled once into a REL file, and used thereafter as a library.

Searches of libraries and the search order can be controlled by the STDIO.H #define switches lib1, lib2, lib3.

For example, suppose that the main program is called V.C and is is to be linked with modules V1.C, V2.C, V3.C, VG.C.

Further, assume that V3.C requires CLIBM.REL during link, but V.C does not. Then V.C should be set up as follows:

```
#define lib1 V1,V2,V3,CLIBM,VG
#include <stdio.h>
main() {
  ... code as usual
}
```

The command line given to make V.COM is CLINK V. All linkage is handled automatically, in the order specified, which is equivalent to:

```
L80 V,V1,V2,V3,CLIBM/S,VG,CLIB/S,V/N/E
```

It is important to note that modules V1, V2, V3, VG are assumed to be compiled separately with V1.REL, V2.REL, V3.REL, VG.REL on the default disk. The linker L80 looks on the default drive for libraries. Sometimes you can omit the CLIBM reference because it already exists in the file V3.REL (due to using STDIO.H).

```
#define ccspace 4096
#include <stdio.h>
main()                    /* this is OVERLAY.COM */
{
char *p,*peekw();
extern char *CC CCP;
     p = peekw(I);
     puts("Overlay running");
     if(strcmp(p-4096,"KeyWord") == 0) {
         puts(p-4096+1+strlen("KeyWord"));   /* display data */
     }
     else {
          puts("Bad overlay linkage"); exit();
     }
}
```

Style dictates that you isolate compiler and library dependent source code from the rest of your program by the use of pre-processor #define switches. Here is what we suggest:

```
#define C80 1              /* C/80 & Library 3.2 switch */

#if C80
#define  nullcmd 1         /* Turn off re-direction */
#define nfiles 3           /* Three file buffers    */
#define bufsz1 4096        /* first buffer size 4k  */
#endif

#include <stdio.h>         /* Now read in STDIO.H   */
```

STDIO.H DEFINITIONS

The file STDIO.H is to be used with M80 assemblies of subroutines and the main program. The following assembly switches are used. Declarations are preprocessor #define directives, which must appear before STDIO.H is read into the source code. We use **boldface** and lowercase to distinguish the user-settable switches. Constants are all UPPERCASE.

stderr **stdin** **stdout**	The Bell Labs Unix symbols are defined in header file STDIO.H to use usual defaults: stderr=252, stdin=fin, stdout=fout. The latter have been implemented as function returns from fin__() and fout__().
TRUE FALSE	Both TRUE and FALSE are defined as in the standard set by Bell Lab's Portable C Compiler (PCC). FALSE = 0, TRUE = 1.
_exit	The standard abort exit. Does not flush file buffers. Does not close the file control block to the disk directory. Leading underlines cause trouble with M80/L80, hence the pre-processor solution that you see here.
bufsz1 **bufsz2** **bufsz3**	Settable buffer sizes began with version 3.0. Three switches are provided in STDIO.H. The default is 256. It is suggested that you use size 4096 for fast I/O on sequential files. For rapid seeks, use size 128 or 256. Usage: **#define bufsz1 4096**, before #include <stdio.h>.
charIo	Use **#define charIo 1** before #include <stdio.h>. Most of the fat for a C object file comes from the file support package and re-direction modules. This switch unloads the fat. Recommended for programs like the Sieve of Eratosthenes, which simply prints to the screen.
ccspace	The space under CP/M can be adjusted at compile time via the switch **#define ccspace nnn**. Choose nnn = 1024 to get 1k of free space, etc. This feature is useful if you wish to leave the CCP resident and write your own exit routine. It also provides for a method of chaining variables without stack collision.
nullcmd	Unload the command line package with **#define nullcmd 1**. Use it before the **#include <stdio.h>** in your source file.

Example. Sample incremental development.

Below you will see a complete diary of a sample program V.COM, created by incremental development via modules V1, V2, V3, VG, V.

```
A>type V1.C
#include <stdio.h>
f()
{
  printf("Function f\n");
}

A>type V2.C
#include <stdio.h>
g()
{
 fputs("Function g\n",stderr);
}

A>type V3.C
#include <vg.h>
#include <stdio.h>
char *h()
{
     ltoa((long)x,buffer);
}

A>type VG.C
/*
 * VG.C
 * globals used by V,V1,V2,V3
 */
char buffer[21];
long x = 3546091;

A>type VG.H
/*
 * VG.H
 * globals used by V.C, V1.C, V2.C, V3.C
 */
extern char buffer[21];
extern long x;

A>type V.C
#define libl V1,V2,V3,CLIBM,VG
#include <stdio.h>
main()
{
  f();
  g();
  h();
}

A>clink v    - This compiles and links the program automatically.
```

This example may work with CLIBM omitted from the list, because STDIO.H inserts a library request into V3.REL, causing a search of the library CLIBM.REL. However, there is no sure way to control the search order except by the method above.

Data statements cannot force a library search. But executable ones that use the reserved symbols in STDIO.H will cause a search. The usual error made by programmers is the ommission of #include <stdio.h> or incremental compiles that do not document the library searches back in main().

MODULES — A module is any C function or source file that does not contain main(). In versions 3.1 or earlier a module declaration was needed. This has been automated in version 3.2 by declaring it a module if the reserved word main does not appear.

mathlib — The switch for a library request to CLIBM.REL is **#define mathlib 1**, used before #include <stdio.h>. Note that the printf and scanf float & long switches will automatically request CLIBM.REL as needed. Further, any explicit use of long or float will cause CLIBM to be searched.

sfFLOAT — Turns on the float support for scanf(). Usage: **#define sfFLOAT 1**, before the normal #include <stdio.h>, enables floating-point code for scanf, fscanf, sscanf.

sfLONG — Turns on the long support for scanf(). If neither are selected but your code contains float and long references, then formats like %ld and %f will be processed but not acted upon. The only warning issued is the unexpected performance.

pfFLOAT — Turns on the float support for printf(). Switches the printf() definitions from the main library version to the portable Unix-standard version. Usage: Before #include <stdio.h>, enter the line **#define pfFLOAT 1.**

pfLONG — Same function as pfFLOAT only for long integer support. You may use either or both or none. Using neither causes the main library printf to be used. To force use of the portable printf, #include the proper header file (but #undef printf first!). Usage: **#define pfLONG** 1.

nfiles — The number of files that may be open simultaneously is a constant MAXCHN in the module IOTABLE.REL. To change it from the usual 6, alter and re-assemble IOTABLE. To change the number of I/O buffers from the usual 3, use **#define nfiles nnn** where nnn is any number from 0 to 6. The number of file buffers and their size affects the total heap space available to the program.

STDIO.H DEFINITIONS

EOF	The Unix standard of -1 for end of file is used throughout the library.
NULL	The usual value of 0 is used for NULL. The cast of this variable is always (int) unless otherwise changed.
FILE	The Unix system uses pointers for stream I/O whereas C/80 uses the older version 5 system standard of channel numbers. To make them compatible in source, define FILE to be (int) as in STDIO.H and use **FILE *chan;** rather than **int chan;** in your source code.
lib1 **lib2** **lib3**	There are three switches provided: lib1, lib2, lib3. These variables are actually C/80 macros used by the pre-processor. The effect is to put .REQUEST lib1 in the source code, etc. You may use any string for the macro replacement (C/80 version 3.0 has a bug, but 3.1 and 2.0 work as expected.). Usage: **#define lib1 mylibrary.**
puts	The version of **puts()** in the main library meets the Unix standard. It supports multiple arguments.
type	The function called **type()** is modeled after the function of the same name used in Decsystem Fortran. It types out strings using putchar(). It differs from puts() in that newlines are not output automatically. This function is defined by a MACRO in stdio.h - you may change the name TYPE to whatever seems appropriate.
error	The function called **error()** is the same as type(), except that it always writes to the console. It is a replacement for printf in ROM-based applications.
conout	The **conout()** code supports multiple-argument printing to the console only. It is used primarily as a device driver for screen graphics, in which the delivery rate must be as rapid as possible.
gets	Unix standard get-string from **stdin.** Strips the linefeed from the string so that **puts()** will output the same thing that you type.

ltoa	Converts long integer to ascii text, similar to itoa(). Located in CLIBM.REL with library select done by STDIO.H. See K&R.
atol	Converts ascii text to a long integer, similar to atoi(). Located in CLIBM.REL with library select done by STDIO.H. See K&R. Object code from the C/80 Mathpack.
ftoa	Converts float to ascii text. Provided with the C/80 MATHPACK from The Software Toolworks. Not a K&R function - this one uses four arguments, not two. Located in CLIBM.REL.
LIB	A macro definition for M80, which inserts library request information into the MAC file.
SETLIB	A macro definition for M80, which sets a flag for use of a particular library. Used to auto-select libraries based on reserved symbol names.
USELIB	A macro definition for M80, which invokes a library request for L80.
ULIB TLIB MLIB XLIB	These assembly constants are used as switches to invoke searches of libraries CLIBU, CLIBT, CLIBM and CLIBX, respectively. See also the header files MATH.H, UNIX,H.

The following keywords are used to set up hooks to the libraries:

SLONG., LLONG., seek, wrDISK, rdDISK, timer, numsort, ssort3, ssort2, dsort, dsort16, waitz, binary, sbinary, run, chain, insert, STK_pos, pokel, peekl, fseek, ftellu, getl, putl, randl, ltoa, ftoa, atof, atol

UNIX.H DEFINITIONS

The header file UNIX.H contains the following definitions.

```
#define _tolower      tolower         /* C/80 has no argument macros */
#define _toupper      toupper
#define fopen         fopena          /* Ascii open */
#define freopen       freopa          /* Ascii re-open */
#define ftell         ftellu          /* Unix ftell */
#define lseek         fseek           /* happen to match in C/80 */
#define read          reada           /* ascii read */
#define write         writea          /* ascii write */
#define fgetc         getc            /* both are functions */
#define open          opena           /* ascii open */
#define cfree         free            /* cfree for calloc */
```

In addition, it inserts into your program a library request for the library CLIBU.REL.

_toupper
_tolower — These conversion macros are not possible under C/80, so we map them to genuine function calls.

fopen — The honest function is fopena(). We also include fopenb() for binary I/O, but name fopen() is assigned to fopena(). Note that the C/80 name fopen() does not conflict. The only possible disaster is in leaving off #include <UNIX.H>, which drops all the hooks. Source code compiled with UNIX.H will port to most Unix V7 systems without change.

freopen — The normal freopen is ASCII mode. To get binary mode you have to use freopb(), which does not have a portable Unix V7 analog.

ftell — Past naming conventions of the C/80 library use ftell() with an integer return. This ftell() has a standard Unix V7 long integer return. The actual function used is ftellu(), but to keep the source code looking like Unix V7, please use ftell().

lseek — It so happens that lseek() exists in the library, by accident. This is an obsolete Unix function, but it may be handy to know how to access it under this library.

cfree — The cfree() function paired to calloc() is identical to the free() function used with malloc().

read write	The naming conventions of C/80 again conflict with a standard Unix V7 interface. The problem is that the Software Toolworks functions operate in fixed binary mode. They are incapable of using devices and will not translate carriage return and linefeed combinations. Worst, there is no real end of file detection. To cure the problem, we created functions reada(), readb(), writea(), writeb() which do real stream I/O. In UNIX.H, we assign read to reada and write to writea. Code written with UNIX.H uses standard names that transport easily to all full C systems.
fgetc	The function fgetc() is identical to getc(). Under most systems, getc() is a macro definition. But C/80 does not support such things, so getc() and fgetc() are one and the same.
open	The honest functions are opena() and openb(), which do ASCII and BINARY opens, respectively. We name open() as opena() to be Unix V7 compatible.

SETJMP.H DEFINITIONS

The header file SETJMP.H contains the following essential lines, which MUST be used if your source code calls either setjmp() or longjmp().

```
struct jmpbuffer { char *STACKP; char *RETADR;};
#define jmp_buf struct jmpbuffer
```

Recall that setjmp and longjmp act as a pair to back out of deeply buried function calls. Suppose, for example, in the process of opening up data files it is found at some level that a data file is corrupt. The function pair setjmp and longjmp give an elegant way to back out of the mess, close the data files and start over.

To make it clear how the pair setjmp() and longjmp() interact in source, we reproduce here the example found in SETJMP.H.

Example: Test program for setjmp() and longjmp().

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf env1[1];
jmp_buf env2[1];

f(fp)
FILE *fp;
{ int x;
 x = getcbinary(fp); puts("");
 switch(x) {
 case 'Z'-'@': longjmp(env1,10);
 case '\n'   : longjmp(env1,20);
 case 'C'-'@': longjmp(env1,30);
 case 'A'-'@': longjmp(env2,1);
 }
}

main() {
   puts("Setup for env1");
   switch(setjmp(env1)) {
    case 0: break;
    case 10: puts("EOF encountered on input");
    case 20: puts("Linefeed in input"); break;
    case 30: puts("Ctrl-C normal exit"); exit();
    default: puts("Undefined error");
   }
   puts("Setup for env2");
   switch(setjmp(env2)) {
     case 0: break;
     case 1: puts("ctrl-A struck"); break;
     default: puts("Undefined error");
   }
   puts("Special characters are ctrl-A, ctrl-C, ctrl-J, ctrl-Z");
   while(TRUE) f(stdin);
}
```

Below is the contents of MATH.H. Please take note that MATHDEF.H is different from MATH.H. The former is used to compile the library CLIBT.REL.

```
#define FLOAT           float
#define PI              (FLOAT)3.14159265359
#define PIbyTWO         (FLOAT)1.570796326795
#define TWObyPI         (FLOAT)0.63661977
#define ONE             (FLOAT)1.0
#define ZERO            (FLOAT)0.0
#define INF             (FLOAT)1.0e39
#define EXPLARGE        (FLOAT)89.80081863
#define POW10INF        (FLOAT)39.0
#define MAX2EXP         129
#define PIby180         (FLOAT)0.0174532925
#define IPIby180        (FLOAT)57.29577951
#define EDOM            33
#define ERANGE          34
#define ln              log
#asm
TLIB    SET     1
#endasm
```

The purpose of the symbol TLIB is to provide linkage to the library CLIBT.REL. This is done by a switch in STDIO.H, therefore it is essential that MATH.H be read into your source file prior to STDIO.H.

ERRNO.H DEFINITIONS

Below are the error codes normally supported under Unix V7 from Bell Labs.

The external variable

extern int errno;

should be declared in any source code that uses the error reporting features of the library. The variable itself is located in CLIB.REL. You may put it in your source code if desired, thereby omitting the library variable during link.

At present, error reports are supported in the library CLIBT.REL only. To print an error, you may use the function perror() in CLIBU.REL, or else write your own.

The error codes that are supported appear in perror.c in the source archives. We refer you to that source plus any standard reference on the Bell Labs Unix System III error codes. A printed copy of perror.c appears in the library manual.

```
/*
 * errno.h - error codes
 */
#define EPERM   1           #define EFBIG   27
#define ENOENT  2           #define ENOSPC  28
#define ESRCH   3           #define ESPIPE  29
#define EINTR   4           #define EROFS   30
#define EIO     5           #define EMLINK  31
#define ENXIO   6           #define EPIPE   32
#define E2BIG   7
#define ENOEXEC 8           /* math bound checks */
#define EBADF   9           #define EDOM    33
#define ECHILD  10          #define ERANGE  34
#define EAGAIN  11
#define ENOMEM  12
#define EACCES  13
#define EFAULT  14
#define ENOTBLK 15
#define EBUSY   16
#define EEXIST  17
#define EXDEV   18
#define ENODEV  19
#define ENOTDIR 20
#define EISDIR  21
#define EINVAL  22
#define ENFILE  23
#define EMFILE  24
#define ENOTTY  25
#define ETXTBSY 26
```

The sources for CLIBT.REL are in the transcendental function archive. All header files required are included. Please note that this library uses special properties of The Software Toolworks MathPack. There is no claim made about the portability of the code. Unix code assumes doubles, and this package uses float only.

The transcendental function library is built from the following command line sequence. Ordering of functions is essential, because the references in the library must resolve in one pass. In addition, a library search of CLIBM.REL is required to pick up the needed parts of the long and float library.

```
B:
A:LIB
CLIBT=
RAND
ACOT
ACOS
ACSC
ASIN
ASEC
ATAN2
ATAN
LDEXP
FREXP
TAN
COS
SIN
SQRT
SINH
COSH
TANH
EXP
POW
POW10
LOG
LOG10
CEIL
FLOOR
FMOD
MODF
RAD
DEG
LABS
FABS
ABS
HORNER
/E
```

LIBRARY SOURCE MANAGER. The main features of ARC.COM are as follows:

- Several versions of the same source can be maintained. Files are distinguished by their physical location in the archive, the date, and finally the actual contents.
- Archives may be combined with PIP to make a master archive. Archives are text files terminated with ctrl-Z.
- Single files can be copied out of the archive. This includes paged printing with settable left margin.
- Archives can be split into smaller ones.
- Files within an archive may be selectively deleted.

The functions of adding files to an archive versus the management of the archive have been separated. ARC.COM handles an archive after creation. MARC.COM creates archives and adds files. Following are the functions of MARC:

- Adding files to an archive uses wildcards from the command line. The speed is similar to PIP, but you can view the file and add notes. Files in the list may be archived or skipped over.
- Many small files can be combined into an archive to make better use of disk space. For example, limitations of 64 directory entries disappear, because the archive uses full blocks.
- The principal reason why MARC exists is to back-up on disk several versions of a C program during development. You don't change the name, just the date and the notes. Usage:

  A>marc myfile.c "-d<today's date>" -aBACKUP.ARC -c

  The best and easiest implementation uses a CP/M submit file.

ARC is capable of producing neat listings on the printer. Just write to the LST: device in SINGLE COPY mode. Options of LEFT MARGIN and PAGE LENGTH can be set, for automatically paging the output and allowing for ring-binder holes. Output to LST: ends with a formfeed.

ARC will tag module names taken from a source file. The result is that files in the archive are tagged, just as though you had typed a tag command T at the console for each file. This feature is very useful for collecting source files into one file.

A sample run of ARC and MARC appears in an appendix. We refer the reader to those pages for detailed information on running the programs.

PORTABLE SOURCE CODE GENERATOR. The generator creates source code files for programs that access the libraries used by L80. In theory, you can generate the complete source required for a ROM application.

The generator is a two-step operation which filters the symbol file from L80 through XREFSYM.COM and finally through ARC.COM to produce the portable source archive.

The purpose of the symbol cross-reference generator called XREFSYM.C is to generate a cross-reference table which maps the global symbols in the REL file to the source code modules in the archives LIBC.ARC and LIBCC.ARC. The information needed by XREFSYM.COM resides in the archive SYMBOLS.ARC and the Intel Symbol file produced by Microsoft's L80.

To operate XREFSYM.COM, for a source file TRIAL.C, do these steps:

```
A>CC TRIAL
A>M80 =TRIAL
A>L80 TRIAL,TRIAL/N/Y/E
A>ERA TRIAL.REL
A>XREFSYM TRIAL.SYM
```

The output is SYMBOLS.SYM, which is used by ARC.COM to tag the relevant files, and make a complete portable source file for the program TRIAL.C.

The condition and readability of the source code that is generated is a function of the source library and its contents. It is suggested that one maintain a **C** - only source library, and forget about trying to transfer the massive I/O library. Generally, files with extension **.CC** are not portable, however exceptions are likely.

A run of XREFSYM is given in an appendix, along with more detail on usage. Generally, XREFSYM is used intermittently. It is little use to casual users.

SETTABLE BUFFER SIZES. New with version 3.0 of the library is the option of settable buffer size and the possibility of dynamic re-assigment of buffers. The changes to the library tightened error checking and added three entries into the I/O table (namely, IOsize, IOnchx, IOend) - see IOTABLE.CC in LIBCC.ARC.

Generally, you should set the buffer size and the number of files by means of macro switches in STDIO.H. However, it is possible to dynamically alter the buffer size and buffer location at run time. This is safe only in case you precede such changes with a rewind() function call on the open stream. See setbuf(), rewind() and the examples therein.

The fixed initial size of each buffer is defined in STDIO.H by the assembly constants $SIZ1, ... ,$SIZ6 . Two other constants $TAG1 and $OFF1 are generated for use in IOTABLE.REL (see STDIO.H). The default size is 256.

The number of files is fixed at 6. Devices CON:, RDR:, PUN:, LST: are recognized but are not part of the file count. They use device numbers 252, 253, 254, 255. Device 0 is mapped to device 252.

Up to 251 files may be in simultaneous use, provided the module IOTABLE.REL is assembled to reselect the correct number of file buffers. See IOTABLE.CC for re-assembly instructions and STDIO.H for header file declarations.

To replace re-assembled IOTABLE.REL in the main library CLIB.REL, use this command line for the Microsoft library manager LIB.COM:

```
A>LIB NEWLIB=CLIB<..IOTABLE>,IOTABLE,CLIB<IOTABLE+1..>/E
A>ERA CLIB.REL
A>REN CLIB.REL=NEWLIB.REL
```

The file buffers and file control blocks for each of the files selected with the nfiles assembly switch (#define nfiles 6, for example) reside physically in memory just after the program data and text.

File buffers and file control blocks built at run time in excess of the nfiles count (but not more than **MAXCHN-1** are possible) will call sbrk() to get more space. The sfree() function that frees up heap space obtained from sbrk() will not free up file buffer areas. A fragmented heap can result. The latter can cause program failure due to lack of continguous heap space.

To manually throw away the buffers assigned to file descriptor fd, use this function (not portable and potentially dangerous):

```
unbuff(fd) int fd;
{
extern char IOch[]; extern int IOfcb[],IObuf[];
     if(IOch[fd] == -1) IOfcb[fd] = IObuf[fd] = 0;
}
```

While the above throws away the buffers, sbrk() will not forget until you call sfree() to reset the end-of-program pointer. Use sfree(-1) to get rid of all the assigned heap space. The latter includes everything sbrk() assigned up to the call to sfree.

FORCE-LOADING OF REL FILES. The file CLIB.REL has the same use as FORLIB.REL in FORTRAN. The STDIO.H file causes modules compiled by C/80 to undergo a library search of CLIB.REL on exit from L80.

Modules other than CLIB.REL can be force-searched by using the STDIO.H symbols lib1, lib2, and lib3. The search order is the order of appearance, with final search of CLIBX, CLIBM, CLIB, in that order, as required.

You will usually not have to force the search of the libraries. The header file STDIO.H causes most of the search requests to be processed automatically.

LIBRARY INTEGRITY CHECK. Below is a sample run of LIB.COM to verify forward references in the library CLIB.REL (use the /U - switch). CLIB.REL must start with its fixed-origin routine CCMAIN.REL and end with END.REL. The latter is used by storage allocation, therefore it must be the last module linked by L80. Your customized library passes the test if it reproduces the sample run.

```
A>LIB
*CLIB/U
Unsatisfied external request(s):
$LM     A EXIT  CBUF    CMODE   CTLB.   EXIT    EX SPC  FIN
FOUT    MAIN    TOT SZ  $OFF1   $OFF2   $OFF3   $OFF4   $OFF5
$OFF6   $SIZ1   $SIZ2   $SIZ3   $SIZ4   $SIZ5   $SIZ6   $TAG1
$TAG2   $TAG3   $TAG4   $TAG5   $TAG6
*^C
A> end sample run

Symbol generation
YOUR PROGRAM: MAIN
CCMAIN.REL:   $LM  A EXIT  CBUF  CMODE  CTLB.  EXIT  FIN  FOUT
STDIO.H:      EX SPC TOT SZ
              $OFF1  $OFF2  $OFF3  $OFF4  $OFF5  $OFF6
              $SIZ1  $SIZ$  $SIZ3  $SIZ4  $SIZ5  $SIZ6
              $TAG1  $TAG$  $TAG3  $TAG4  $TAG5  $TAG6
```

MAINTENANCE AND RE-BUILDS. To replace module sample in CLIB.REL, use Microsoft's LIB.COM as follows:

```
A>LIB NEWLIB=CLIB<sample-1..>,sample,CLIB<sample+1..>/E
A>ERA CLIB.REL
A>REN CLIB.REL=NEWLIB.REL
```

To extract a module from CLIB.REL:

```
A>LIB SAMPLE=CLIB<sample>/E
```

To list the global symbols in CLIB.REL:

```
A>LIB CLIB/L
```

DATA TYPES. The supported data types and their sizes are:

char	8 bits
short	16 bits
int	16 bits
unsigned	16 bits
long	32 bits
float	32 bits

There is no double data type. To convert existing programs that use doubles, or to develop new programs that will ultimately use doubles, place in your program:

```
#define double float
```

LONG INTEGER FORMAT. The format for long integers is reverse intel format, so that an integer is the lower 16 bits of a long integer. Long and integer pointers will point to the lowest byte. There is no unsigned long data type at present, however you may simulate it and print it out using printf().

FLOAT EXPONENT RANGE. Floats may have exponents in the range -38 to +38. The exponent format is 128-complement in the last Intel reverse format byte of the 32-bit word. See the sources in CLIBT.ARC for detailed information, especially fdexp() and frexp().

CALLING CONVENTIONS - C FUNCTIONS.

The arguments for a function call are pushed onto the stack in the order listed. On return, the arguments are popped off the stack by the most efficient method.

The stack adjustment does not affect the return value, which is HL in the case of integer returns or a dual-register pair for long or float returns.

All stack pushes are 16 bits or 32 bits. Arguments that are long integer or float values cause a 32-bit push.

To unscramble a function call in assembly language, POP HL to obtain the return address. Each successive POP yanks one argument of width 16 bits, in the reverse order of the argument list. For example,

```
int x;
int f(a,b)
int a,b;
{
     x = a; x = b;
}
```

is written in assembler as:

```
     DSEG
x::  DW   0          ;storage is global, in data segment
     CSEG
f::  DS   0          ;function f is a global, in code segment
     POP HL          ;return address of caller
     POP DE          ;value of argument b
     POP BC          ;value of argument a
     PUSH BC
     PUSH DE
     PUSH HL         ;stack restored, ready to use DE,BC
     LXI  HL,x
     MOV  M,C
     INX  H
     MOV  M,B        ;x = a
     LXI  HL,x
     MOV  M,E
     INX  H
     MOV  M,D        ;x = b
     RET
```

There are some common tricks used in the C/80 assembly interface, often seen in C source code. One such is:

```
exitto(addr) char *addr;
{
     addr;
#asm
     SPHL
#endasm
}
```

The purpose of the strange "addr;" in the function body is to load register pair HL with the address in the argument list.

The assembly code immediately following the statement loads the stack pointer with HL and the implied return effects a processor return from the stack at the aforementioned address. In 8080 code:

```
exitto::  DS 0
          LXI H,2         ;fetch addr off stack
          DAD SP
          CALL h.##       ;load HL indirect from HL
          SPHL            ;this is the assembly code you see above
          RET             ;ending brace is a RETURN
```

Such code as above is generally useless, unless you had the foresight to save the caller's stack frame and make your own stack. The possibility of corrupting the caller's stack still looms in the background, so we leave the issue with the programmer.

MEMORY LAYOUT UNDER CP/M 2.2. The standard C/80 configuration is altered somewhat in this library, to allow for dynamic file buffers. The test program MEMMAP.C illustrates how:

```
#include <stdio.h>
#define CPMBASE         0       /* cpm base */
#define TBUFFER         0x80    /* transient buffer */
#define CPMSTART        0x100   /* start address transient program */
main(argc,argv) int argc; char **argv;
{
int i;
auto int k;
extern int IOfcb[],IObuf[];
extern char *Cbuf;
extern char *p_fin,*p_fout;
char *sbrk();
     if(argc <= 1) {
          puts("MEMMAP displays the memory map of a C program");
          puts("including command line arguments and buffers.");
          puts("Usage: A>memmap arg1 arg2 ...");
          exit(0);
     }
     fprintf(stderr,"CP/M base = %04x Hex\n",CPMBASE);
     fprintf(stderr,"CP/M default buffer = %04x Hex\n",TBUFFER);
     fprintf(stderr,"CP/M program start address = %04x Hex\n",CPMSTART);

     fprintf(stderr,
            "Physical end of code and data = %04x Hex\n",IOfcb[1]);
     for(i=1;i<=nfiles;++i) {
          fprintf(stderr,"IOfcb[%d] = %04x Hex\n",i,IOfcb[i]);
          fprintf(stderr,"IObuf[%d] = %04x Hex\n",i,IObuf[i]);
     }

     fprintf(stderr,"Heap starts at %04x Hex\n",sbrk(0));
     fprintf(stderr,"Top of heap is at stack = %04x Hex\n",&k);
     fprintf(stderr,"Token table starts at %04x Hex\n",&argv[0]);
     fprintf(stderr,
            "Command line copy starts at %04x Hex\n",&argv[0][0]);
     fprintf(stderr,"Console buffer starts at %04x Hex\n",Cbuf);
     fprintf(stderr,"Console buffer ends at %04x Hex\n",Cbuf+140);
     fprintf(stderr,"Base of CP/M FDOS is at %04x Hex\n",peekw(6));
     fprintf(stderr,
            "C program upper limit is FDOS - ccspace = %04x Hex\n",
                              peekw(6) - ccspace);
     fprintf(stderr,"Indirection file name for >: %s\n",
                    p_fout ? p_fout : "NONE SUPPLIED");
     fprintf(stderr,"Indirection file name for <: %s\n",
                    p_fin ? p_fin : "NONE SUPPLIED");
     fprintf(stderr,"Program name: %s\n",argv[0]);
     for(i=1;i<argc;++i) {
        fprintf(stderr,"argv[%d] = %s\n",i,argv[i]);
     }
}
```

SAMPLE RUN OF MEMMAP.C

Here is a sample run. Note the mapping of lower to upper by the CCP and the handling of quoted characters.

```
A>memmap arg1 arg2 >foo <memmap.com "white space test" arg3
CP/M base = 0000 Hex
CP/M default buffer = 0080 Hex
CP/M program start address = 0100 Hex
Physical end of code and data = 106C Hex
IOfcb[1] = 106C Hex
IObuf[1] = 1090 Hex
IOfcb[2] = 1190 Hex
IObuf[2] = 11B4 Hex
IOfcb[3] = 12B4 Hex
IObuf[3] = 12D8 Hex
Heap starts at 13D8 Hex
Top of heap is at stack = D5E9 Hex
Token table starts at D5F1 Hex
Command line copy starts at D635 Hex
Console buffer starts at D676 Hex
Console buffer ends at D702 Hex
Base of CP/M FDOS is at D706 Hex
C program upper limit is FDOS - ccspace = D706 Hex
Indirection file name for >: FOO
Indirection file name for <: MEMMAP.COM
Program name: MEMMAP
argv[1] = ARG1
argv[2] = ARG2
argv[3] = WHITE SPACE TEST
argv[4] = ARG3

A>
```

```
/*
 * UNIXIO.H
 *
 * Purpose:
 *
 *      Provides stub functions for unix system calls and
 *      unix functions that have have no analog under CP/M.
 *
 * Usage:
 *
 *      Copy the essential portions of this source file into
 *      your source code. Delete all unwanted lines.
 *
 * Example: The cure for MULTIPLY-DEFINED GLOBAL.
 *
 *      #include <unix.h>
 *      #include <stdio.h>
 *      monitor() { return -1;}
 *      main() {
 *              alarm(1);
 *              wait(2);
 *      }
 *      wait(x) int x;
 *      {
 *              waitz(500*x);
 *      }
 *
 *      The above code will end up with WAIT as a MULTIPLY-DEFINED
 *      GLOBAL. WAIT is already a global. ALARM selects the module
 *      UNIXIO.REL from CLIBU and hence a second instance of WAIT.
 *
 * The Cure:
 *
 *      Edit the source, pasting in as much of this file as required
 *      to resolve all symbols within the source files. To wit, add:
 *
 *                      alarm() { return 0;}
 *
 *
 * Basic Reference: Banahan & Rutter, The Unix Book, Wiley Press,
 *                  New York (1983).
 *
 * Unix Reference: The Unix Programmers Manual
 *
 *
 * ===UNIX SYSTEM CALLS====
 *
 * int acct(file)                       Return 0.
 * char *file;
 *
 * unsigned alarm(seconds)              Return 0.
```

```
* unsigned second;
*
* char *cuserid(s)                  Return (char *)0.
* char *s;
*
* int dup(old)                      Return -1 for error.
*
* int dup2(old,new)                 Return -1 for error.
*
* int fork()                        Return -1 for error.
*
* #include <sys/types.h>
* #include <sys/stat.h>
* int fstat(fd,buf)                 Return -1 for error.
* int fd;
* struct stat *buf;
*
* #include <sys/types.h>
* #include <sys/timeb.h>
* void ftime(tp)                    Void return.
* struct timeb *tp;
*
* int getpid()                      Return 0.
*
* int getuid()                      Return 0.
*
* int getgid()                      Return 0.
*
* int geteuid()                     Return 0.
*
* int getegid()                     Return 0.
*
* int ioctl(fd,request,argp)        Return -1 for error.
* int fd,request;
* struct sgtty *argp;
*
* int getpgrp()                     Return 0.
*
* int getppid()                     Return 0.
*
* int kill(pid,sig)                 Return -1 for error.
* int pid,sig;
*
* int link(path1,path2)             Return -1 for error.
* char *path1,*path2;
*
* int mknod(path,mode,dev)          Return -1 for error.
* char *path;
* int mode,dev;
*
* int mount(spec,dir,rwflag)        Return -1 for error.
* char *spec,*dir;
```

```
* int rwflag;
*
* int nice(incr)                        Return 0.
* int incr;
*
* void pause()                          Void return.
*
* int pclose(fp)                        Return -1 for error.
* FILE *fp;
*
* int pipe(fd)                          Return -1 for error.
* int fd[2];
*
* FILE *popen(command,type)             Return 0.
* char *command,*type;
*
* void profil(buff,bufsize,offset,scale)        Void return.
* char *buff;
* int bufsize,offset,scale;
*
* int ptrace(request,pid,addr,data)     Return -1 for error.
* int request,pid,addr,data;
*
* int putpwent(p,fp)                    Return -1 for error.
* struct passwd *p;
* FILE *fp;
*
* int setgid(gid)                       Return -1 for error.
* int gid;
*
* int setpgrp()                         Return 0.
*
* int setuid(uid)                       Return -1 for error.
* int uid;
*
* #include <sys/types.h>
* #include <sys/stat.h>
* int stat(name,buf)                    Return -1 for error.
* char *name;
* struct stat *buf;
*
* void sync()                           Void return.
*
* long time(tloc)                       Returns (long)0
* long *tloc;
*
* #include <sys/types.h>
* #include <sys/timeb.h>
* long times(buffer)                    Return -1 for error.
* struct tbuffer *buffer;
*
* int stime(tp)                         Return -1 for error.
```

```
* long *tp;
*
* int umask(cmask)                    Return 0.
* int cmask;
*
* int umount(spec)                    Return -1 for error.
* char *spec;
*
* int uname(name)                     Return 0.
* struct utsname *name;
*
* char *userid(s)                     Return (char *)0
* char *s;
*
* int ustat(dev,buf)                  Return -1 for error.
* int dev;
* struct ustat *buf;
*
* #include <sys/types.h>
* int utime(path,times)               Return -1 for error.
* char *path;
* struct utimbuf *times;  or  time_t timep[2];
*
* int wait(stat_loc)                  Return -1 for error.
* int *stat_loc;
*
*
* ===UNIX MISCELLANY===
*
* #include <grp.h>
* struct group *getgrent()            Return 0.
*
* #include <grp.h>
* struct group *getgrgid(gid)         Return 0.
* int gid;
*
* #include <grp.h>
* struct group *getgrnam(name)        Return 0.
* char *name;
*
* int setgrent()                      Return 0.
*
* int endgrent()                      Return 0.
*
* char *getlogin()                    Return (char *)0.
*
* int getpw(uid,buf)                  Return -1 for error.
* int uid;
* char *buf;
*
* #include <pwd.h>
* struct passwd *getpwuid(uid)        Return 0.
```

```
 * int uid;
 *
 * #include <pwd.h>
 * struct passwd *getpwent()          Return 0.
 *
 * #include <pwd.h>
 * struct passwd *getpwnam(name)      Return 0.
 * char *name;
 *
 * int setpwent()                     Return 0.
 *
 * int endpwent()                     Return 0.
 *
 * int monitor(lowpc,highpc,buffer,bufsize,nfunc)    Return 0.
 * int (*lowpc)();
 * int (*highpc)();
 * short buffer[];
 * int bufsize;
 * int nfunc;
 *
 * char *ttyname(fd)                  Return (char *)0.
 * int fd;
 *
 * int sleep(seconds)                 Return 0.
 * unsigned seconds;
 */
#asm
times:: DS 0            /* long return */
        LXI B,0         /* return (long)0 */
        LXI D,0
        RET
acct::    DS 0
alarm::   DS 0
getpid::  DS 0
getuid::  DS 0
getgid::  DS 0
geteuid:: DS 0
getegid:: DS 0
getpgrp:: DS 0
getppid:: DS 0
nice::    DS 0
popen::   DS 0
setpgrp:: DS 0
umask::   DS 0
uname::   DS 0
getgrent::DS 0
getgrgid::DS 0
getgrnam::DS 0
setgrent::DS 0
endgrent::DS 0
getpwuid::DS 0
getpwent::DS 0
```

```
getpwnam::DS 0
setpwent::DS 0
endpwent::DS 0
monitor:: DS 0
sleep::   DS 0
cuserid:: DS 0
getlogin::DS 0
        LXI H,0
        RET
dup::     DS 0
dup2::    DS 0
fork::    DS 0
fstat::   DS 0
ioctl::   DS 0
kill::    DS 0
link::    DS 0
mknod::   DS 0
mount::   DS 0
pclose::  DS 0
pipe::    DS 0
ptrace::  DS 0
putpwent::DS 0
setgid::  DS 0
setuid::  DS 0
stat::    DS 0
stime::   DS 0
umount::  DS 0
ustat::   DS 0
utime::   DS 0
wait::    DS 0
getpw::   DS 0
ftime::   DS 0
pause::   DS 0
profil::  DS 0
sync::    DS 0
        LXI H,-1
        RET
        END
#endasm
```

STDIO.H CONTENTS

```
/*STDIO.H vers 3.2*/
#define EOF -1                          End of file
#define NULL 0                          Unix NULL
#define TRUE 1                          Unix TRUE
#define FALSE 0                         Unix FALSE
#define FILE int                        Unix-compatible FILE
#define void int                        Unix-compatible void return
#define VOID int
#define stdin (FILE *)fin__()           Std input stream
#define stdout (FILE *)fout__()         Std output stream
#define stderr (FILE *)252              Std error reporting stream
#define _exit a_exit                    Hook _exit() to real routine
#ifdef chario                           Set up character-only I/O
putchar(){
#asm
 MVI A,252                              Console device handle
 JMP $PCH##                             Vector to console output.
#endasm
}
getchar(){
#asm
 JMP $B8##                              Vector to console input.
#endasm
}
#endif
#ifndef pfLONG                          Set up switches for printf.
#ifndef pfFLOAT                         First one is fast, small integer
#define printf prtf_1(),prtf_2          only version in CLIB.REL, which
#define fprintf fprt_1(),fprt_2         has limited recursion facility.
#define sprintf sprt_1(),sprt_2
#endif
#endif
#ifndef printf                          Otherwise, use the portable
#define printf prnt_1(),prnt_2          printf in CLIBM, which has all
#define fprintf prnt_1(),prnt_3         features including long and float.
#define sprintf prnt_1(),prnt_4
#endif
#define puts type_1(),type_3            Define multi-argument puts().
#define type type_1(),type_2            Define puts() without newline.
#define error err_1(),err_2             Alternate type() for console only.
#define conout cono_1(),cono_2          Direct console output, multi-arg.
#define scanf STK_pos(),scan_f          Define scanf for multi-arguments.
#define sscanf STK_pos(),s_scan         Note lack of recursion.
#define fscanf STK_pos(),f_scan
#ifndef ccspace                         Define space above C program if
#define ccspace 0                       not defined by user.
#endif
#ifndef nfiles                          Define number of active files if
#define nfiles 3                        not already defined by user. High
#endif                                  number is 6.
#asm
```

```
#ifdef pfFLOAT                      Set up portable printf switches
#ifndef pfLONG                      for long and float code includes.
 EXT PF_OP1 ;long off               Allows end-user programs with the
#endif                              portable printf and all excess
#endif                              bulk removed.
#ifdef pfLONG
#ifndef pfFLOAT
 EXT PF_OP2 ;float off
#endif
#endif
#ifdef sfLONG                       Set up scanf switches for long
#ifdef sfFLOAT                      and float code includes. These
 EXT SF_OP1 ;long&float             switches allow end-user programs
#else                               with long and/or float bulk
 EXT SF_OP2 ;long only              removed to optimize code size.
#endif
#endif
#ifdef sfFLOAT
#ifndef sfLONG
 EXT SF_OP3 ;float only
#endif
#endif
#ifdef nullcmd                      Set up null command line options
 CSEG                               which get rid of re-direction bulk.
 IF1
 .PRINTX * Re-defined Copen_, Cexit_, tokens, C_mcln *
 ENDIF
Copen_:: DS 0                       Make vacant routines to take the
Cexit_:: DS 0                       place of the real ones.
tokens:: DS 0
C_mcln:: DS 0
 JMP RN$##                          Return 0.
#endif
LIB MACRO Y                         Define library request macro for
 IRP X,<Y>                          Microsoft M80/L80.
 .REQUEST X
 IF2
 .PRINTX * X Request *
 ENDIF
 ENDM
 ENDM
SETLIB MACRO X,Y                    Define set library macro for
 IFDEF X                            Microsoft M80/L80. Turns on a
Y SET 1                             flag which tells us to request
 ENDIF                              a library.
 ENDM
USELIB MACRO X,Y                    Define library use macro for
 IFDEF X                            Microsoft M80/L80. Inserts a
 LIB Y                              library request if the symbol
 ENDIF                              is defined.
 ENDM
```

SETLIB SLONG.,MLIB	Store long integer
SETLIB LLONG.,MLIB	Load long integer
SETLIB seek,XLIB	C/80 seek
SETLIB wrDISK,XLIB	Direct-disk read
SETLIB rdDISK,XLIB	Direct-disk write
SETLIB timer,XLIB	H89/Z90/Z100 timer
SETLIB numsort,XLIB	Distribution sort
SETLIB ssort3,XLIB	Shell sort, MBASIC rules
SETLIB ssort2,XLIB	Shell sort, array
SETLIB dsort,XLIB	Distribution sort
SETLIB dsort16,XLIB	Distribution sort, 16-bit
SETLIB waitz,XLIB	Wait for Z90/Z100
SETLIB binary,XLIB	Binary search, integer array
SETLIB sbinary,XLIB	Binary search, string array
SETLIB run,XLIB	Run CP/M command line
SETLIB chain,XLIB	Chain to tpa, fcb
SETLIB insert,XLIB	Insert CP/M command tail
SETLIB STK_pos,XLIB	Scanf function
SETLIB pokel,MLIB	Poke long integer
SETLIB peekl,MLIB	Peek long integer
SETLIB fseek,MLIB	Long seek, Unix
SETLIB ftellu,MLIB	Long ftell, Unix
SETLIB getl,MLIB	Get long integer from stream
SETLIB putl,MLIB	Put long integer to stream
SETLIB randl,TLIB	Random number primitive
SETLIB ftoa,MLIB	Float to ascii
SETLIB atof,MLIB	Ascii to float
SETLIB ltoa,MLIB	Long to ascii
SETLIB atol,MLIB	Ascii to long
SETLIB prnt_l,MLIB	Portable printf function
SETLIB PF_OP1,MLIB	Printf option
SETLIB PF_OP2,MLIB	Printf option
SETLIB SF_OP1,MLIB	Scanf option
SETLIB SF_OP2,MLIB	Scanf option
SETLIB SF_OP3,MLIB	Scanf option
SETLIB TLTB,MLIB	Scanf option

```
 IFDEF main                          If main program, then insert
 EXT $LM                             hook to link in CCMAIN.REL.
EX_SPC SET ccspace                   Set up extra space under BDOS
TOT_SZ SET 0                         Iterate to compute total fcb
 PUBLIC EX_SPC,TOT_SZ                size
#ifdef bufsz1                        Set up default file buffer sizes.
$SIZ1 SET bufsz1                     Unix I/O should use 512 default.
#endif                               C/80 default is 256.
#ifdef bufsz2
$SIZ2 SET bufsz2
#endif
#ifdef bufsz3
$SIZ3 SET bufsz3
#endif
 IRP X,<1,2,3,4,5,6>                 Compute file buffer+file control
 IFNDEF $SIZ&X                       block sizes and grand total.
$SIZ&X SET 256
 ENDIF
$TAG&X SET -(X LE nfiles)            Set up tag for which ones used.
$OFF&X SET TOT_SZ
TOT_SZ SET $TAG&X*(36+$SIZ&X)+TOT_SZ
 PUBLIC $TAG&X,$SIZ&X,$OFF&X
 ENDM
 ENDIF
#ifdef lib1                          Invoke user library macros.
 LIB <lib1>
#endif                               Usage is like
#ifdef lib2                              #define lib1 V1,V2,V3
 LIB <lib2>                          entered prior to reading in
#endif                               STDIO.H.
#ifdef lib3
 LIB <lib3>
#endif
#ifdef mathlib                       Invoke user math library switch.
MLIB SET 1
#endif
 USELIB ULIB,CLIBU                   Unix library
 USELIB TLIB,CLIBT                   Transcendental function library
 USELIB XLIB,CLIBX                   Main library extension
 USELIB MLIB,CLIBM                   Math library
 USELIB main,CLIB                    Main library
 IF1                                 Print message on pass 1 of M80
 .PRINTX * STDIO.H - Ver 3.2 *       to tell user we're here and the
 ENDIF                               C/80 compiler was used.
#endasm
```

"CON:","LST:","PUN:","RDR:" ... 180,182,212,282
"r" mode ... 94,95,98,104,105
"r+","w+","a","a+" file modes ... 94
"rb" binary file mode ... 94
"u" update file mode ... 94,95
"ub" update binary file mode ... 94
"w" file mode ... 94,95,98,104,105
"wb" write binary file mode ... 94
$DIR ... 5,6,42,43,116,235
$END,$END## ... 25,34,226
$END##+TOT SZ## ... 25,226
$LM ... 34
$R/O,$R/W,$SYS ... 5,6,42,43,116,235
$SIZ1,$SIZ6 ... 146
$STRCV ... 290
%% ... 190,227,284
%b printf binary output ... 35,194
&-operator ... 75
'con:rdr:pun:lst:' ... 145
(*cmp)() ... 26,27,137,161,162,203,205,241
(fscanf(fp,"%d",&d)) return value kludge ... 108
(int)x.c[3]) float exponent access ... 107,155
+INF ... 8,9,10,12,272
-128L ... 111
-32767 ... 15,163
-INF ... 8,9,10,12,56,156,157,188,245,272,274
-PI ... 9,11,12,13
-R-XR-XR-X ... 42
-RWXRWXRWX ... 42
04000 and similar octal access modes ... 42,43
0330 (H89 port) ... 142,183
0x5c,0x6c,0x80 default buffers ... 62,63,88,223
10mhz ... 170
128-byte ... 38,145
1281 ... 110
136-character ... 35
16-bit ... 15,49,67,136,142,159,177,178,187,202,208,290
16777216 ... 111,113
252,253,254,255 device handles ... 78,95,145,211,294
256*SIZBLOCK ... 166
256-byte ... 112
2n+512 distribution sort ... 177
32-bit ... 14,16,17,23,49,68,100,114,129,152,163,187,200,247
32767 signed int limit ... 15,16,54,111,152,163,178,225,226,260
36-byte ... 18,31,164
4-byte ... 103,165
4096 ... 115,236
4mhz clock speed ... 45,178
65535 unsigned limit ... 208,240
65536*256 ... 111,113
68000 CPU ... 129,136,139,170,200,202,270
8-bit ... 142,187

80186,80286 CPU ... 270
8080 CPU ... 15,68,139,152,174,178,270
8086 CPU ... 139,176,270
8088 CPU ... 176,270
8250 UART ... 142,183
8253 CTC ... 142,183,291
<CTYPE.H> ... 150
<GRP.H> ... 288
<MATH.H> ... 7
<PWD.H> ... 288
<SYS/STAT.H> ... 285,287
<SYS/TIMEB.H> ... 285,287
<SYS/TYPES.H> ... 285,287
_exit() ... 1
a_exit() ... 1,34,60
abort(code) ... 2
abs(x) ... 4
access(name,mode) ... 5
acct(file) ... 285
accuracy ... 273,291
acos(x) ... 7
acot(x) ... 8
acsc(x) ... 9
alarm(seconds) ... 285
algorithm ... 177,252
analog-to-digital ... 142,183
approximation ... 157,244,273
archive ... 116,235
asec(x) ... 10
asin(x) ... 7,11
atan(x) ... 12
atan2(x,y) ... 13
atof(s) ... 7,14
atoi(s) ... 4,15
atol(s) ... 16
attribute ... 6,43,116
auto-repeat ... 22
auto-switching ... 174
Banahan & Rutter ... 25,204,226,243,273,285
BASIC Language ... 17,23,30,32,51,144,228,230,285
bdos(code,de) ... 17
BDS-C ... 17,23,175
bell ... 199
benchmark ... 85,275
Bilofsky, Walt ... 146,232
binary(x,v,n) ... 19
bios(code,bc) ... 23,209,292
bitmap ... 234
blank-filled ... 164
blm ... 127
boot+0x5c,boot+0x6c ... 143
boot+0x80 ... 88,89

Bourne ... 141,222
brk(address) ... **24**,103,166,168,216,226
bsearch(key,base,n,w,cmp) ... **26**,162
bsh ... 127
buffered ... 22,35,79,82,94,142
bufsz1,bufsz2,bufsz3 ... **146**,220
byte-sex ... 270
C&H (Cheney & Hart) ... 12,157,188,244,248
C_cmln() ... 34,62
calloc(m,s) ... **28**,103
Cbuf ... **34**,59,143,153
CC CCP ... **34**
ccBDOS(DE,BC) ... **30**,89,128,158
ccBIOS(de,bc,f) ... **32**,128,133
ccMAIN() ... **33**,34
CCP ... 62,143,223,271,279
CCtlCk ... 22,35,45
ceil(f) ... **36**
CEXIT.CC ... 34
Cexit () ... **34**,37,53,72,78,279
cfree(a) ... 29,103
cfs(file) ... **38**
chain(tpa,program) ... **39**,143,271,279,
chdir(s) ... **41**
checksum ... 127,128
Cheney ... 12,244,273
chmod(name,mode) ... **42**
CHmode(n) ... **44**,121,182
chown(name,owner,group) ... **46**
CI-86 ... 175
clearerr(fp) ... **47**
clock ... 183,208,275,291
close(fd) ... **48**,57,179,181
CMCLN.CC ... 34
Cmode ... **21**,22,34,35,45,59,60,118
cmp(key,tbl),cmp(p,q) ... 26,161,242
cmpf(n,m),cmpi(n,m),cmpl(n,m),cmps(n,m) ... **49**,50,204
Cody ... 244
coefficients ... 244
comp(str1,str2) ... 51
con(c) ... **52**
concurrency ... 40
conflush() ... 153
conout(c1,c2,...) ... 52
conversions ... 227,228,244
COPEN.CC ... 34
Copen () ... 34,53,279
core(n) ... **54**,168,240,257,258
cos(u)/sin(u) ... 273
cos(x) ... **55**
cosecant ... 9
cosh(x) ... **56**

cosine ... 7,55,56
cotangent ... 8
CP/M-68k ... 17,23,129,136,139,200,202,209,292
CP/M-80 ... 6,27,76,111,113,209,216,292
CPMEOF ... 220
creat(name,pmode) ... 57,79
creata(name,pmode) ... 57
creatb(name,pmode) ... 57
cross-check ... 209
CtlB,CtlB. ... 34,58
Ctlb () ... 34,45,58,59,60
CtlCK() ... 35,58,59,60
ctrl-b ... 35,45,58,59,60,121,214
ctrl-c ... 35,45,60,238
ctrl-l ... 78
ctrl-p ... 45
ctrl-s ... 45
ctrl-z ... 21,35,,83,120,121,123,135,195,201,211,232,233,294
CTYPE.H ... 151,277
cvupper(s) ... 61,133,141,222,224,256,262,281
DDT ... 2,3,143
debugger ... 2,3
decodF(fcb) ... 63,88,89,164
decodtokens(table) ... 278
deg(x) ... 64,244,273
degree ... 55,64,140,207,244
dfreep->avail,dfreep->bpsec,dfreep->spg,dfreep->total ... 128
direct-disk ... 209,292
dma ... 17,18,23,30,31,32
Donald Knuth ... 65,66,177
double-# ... 25,226
dpbp->dsm,dphp->alvp,dphp->hdpbp ... 127,128
Dr.Dobbs Journal ... 209
DRI ... 17,23
DRIVER.PRE ... 39
drm,dscr[3],dsm ... 127
dskreset(drive) ... 218
dsort(table) ... 65
dsort16(table) ... 67,162
dup(old) ... 285
dup2(old,new) ... 285
EBASE ... 73
EBCDIC ... 276
ECHO.C ... 201
edata ... 25,69,226
EDOM ... 7,10,11,13,156,157,188,248,272
end,END.CC ... 34,69,146
end-of-file ... 297
endgrent() ... 288
endpwent() ... 288
env[1][1] ... 71,237,239
environment ... 237,239

EOS ... 230
eprom ... 71
ERANGE ... 12,56,73,188,245,272,274
Eratosthenes, Sieve of ... 85
errno ... 70,91,107,155,173,185,207,244,272
error(str1,str2,...) ... 71
etext,ETEXT.C ... 25,69,226
EX_SPC ... 34
exit() ... 34,72
exm ... 127
exp(x) ... 56,73,245,274
expand(&argc,&argv) ... 74
EXPLARGE ... 56,245,274
exponent ... 14,73,155,173,188,189
extent ... 194
fabs(value) ... 77,273
fcb[36] ... 164,175
fclose(fp) ... 42,78,84,96,102,160,172,195,231,257,276,284,295
fd=fileno(fp) ... 212,213,295,296
fdopen(fd,mode) ... 79
FDOS ... 40
feof() ... 80,102,119,121,129,136
ferror(fp) ... 81
fflush(fp) ... 48,82,145
fgets(s,n,fp) ... 5,83,230,257,284
fileno(fp) ... 28,80,84,95,104,119,148,168,175,196,282
fileprint() ... 58
fillchr(dest,c,n) ... 85,292
fin,fout ... 34,86
fin__(),fout__() ... 86,87,253
findfirst() ... 76,88,89,217
findnext() ... 76,88,89,217
fixfile(filename,pattern) ... 90
floor(f) ... 91
fmod(x,y) ... 92
fnb(s) ... 93,246
fopen (file,mode) ... 94,95
fopen(name,mode) ... 5,42,96,175,195,200,257,276,284,295
fopena(name,mode) ... 96
fopenb(name,mode) ... 98,213
fork() ... 285
fout__() ... 86,87,253
fprintf(fp,control,arg1,arg2,...) ... 100,212,213,231,253
fputs(s,fp) ... 5,101,201
fread(buff,s,n,fp) ... 102,211
free(p), cfree(p) ... 103,165
freespace(d) ... 125
freopa(name,mode,fp) ... 79,104
freopb(name,mode,fp) ... 104,105
freopen(name,mode,fp) ... 104,106,118,279,284
frexp(x,nptr) ... 107
fscanf(fp,control,&arg1,&arg2,...) ... 108,109

fscanf return kludge ... 109
fseek(fp,offset,position) ... **110**,145,221,232,284
fstat(fd,buf) ... 285
ftell(fp),ftellr(fp) C/80 ... **112**
ftell(fp),ftellu(fp) Unix ... **113**
ftime(tp) ... 285
ftoa(how,pr,f,s) ... 14,**114**
fwrite(buff,s,n,fp) ... **115**,294
GC BIN ... 21,22,118
getatt(name) ... **116**
getbyte() ... 22,35,**118**,153,214
getc(fp)) ... 98,110,**119**,145,211,220,276,283
getcbinary(fp) ... 22,**120**,214,238
getchar() ... 60,82,105,**118**,121,277
getddir(s,p,nmax) ... **122**,123
getdfree(drive,dfp) ... 125,**126**,127
getegid() ... 286
geteuid() ... 286
getgid() ... 46,285
getl(fp) ... **129**,200
getline(s,n) ... 130,219
getln(s,n) ... **131**
getlongs(fp) ... 129
getmem() ... 103,165
getmypassword() ... 133
getpass(q) ... **133**
getpgrp() ... 286
getpid() ... **134**,285
getppid() ... 286
getpw(uid,buf) ... 288
gets(s) ... 75,93,**135**,246
getuid() ... 46,285
getw(fp) ... **136**
getwords(fp) ... 136
graphics ... 52,142,149,183
H89 ... 275,291
handles ... 122,278
Harbison & Steele ... 29,122,138,141,190,222
Hart & Cheney ... 12,244,273
heaps(base,n,cmp) ... 27,**137**,162
heapsize ... 166
heapsort ... 137
Heath/Zenith ... 2,3
highmem() ... **139**,168
Horner(x,p,n) ... **140**
Horner's method ... 140
Hyperbolic ... 245,274
i-node ... 42
IBM ... 276
index(str,c) ... **141**
initsystem() ... 71
inport(port) ... **142**

insert(tail) ... 143
instr(a$,b$) ... 51
instr(n,str1,str2) ... 144
INTEL ... 85,129,136,142,183,200,202,270
interrupt ... 183,243
interrupt-driven ... 35
IObin[maxchn] ... 145,233
IObuf[fd] ... 145,147,240
IOBYTE ... 21
IOch[maxchn] ... 37,95,145,240
IOctl(fd,request,argp) ... 286
IOdev ... 145
IOend[maxchn] ... 145
IOfcb[maxchn] ... 145,147,175,240
IOind[maxchn] ... 145,233
IOmode[maxchn] ... 80,145
IOnchx[maxchn] ... 145
IOpchan[4] ... 145
IOpeof[4] ... 145
IOpread[4] ... 145
IOpwrit[4] ... 145
IOrw[maxchn] ... 145
IOsect[maxchn] ... 145,211,294
IOsize[maxchn] ... 145,147,236
IOTABLE ... 145
IOTABLE.CC ... 69,95,146,147,233
IOtmp ... 145
isalnum(c) ... 150,151
isalpha(c) ... 150,151
isascii(c) ... 150,151
isatty(fd) ... 148
iscntrl(c) ... 150,151
isdigit(c) ... 150,151,230
isgraph(c) ... 149
islower(c) ... 150,151
isprint(c) ... 149,150,151
ispunct(c) ... 150,151
isspace(c) ... 93,150,151,246
isupper(c) ... 150,151
isxdigit(c) ... 150,151
itoa(number,s) ... 152,290
jmpbuffer,jmp_buf ... 71,237,238,239
joysticks ... 142,183
K&R (Kernighan & Ritchie) ... 103,114,120,132,150,166,192,194,197
Kernighan ... 242
keyboards ... 22
keystat() ... 153
kill(pid,sig) ... 286
kludge (multi-arg) ... 109,182,194,230,250
Knuth, Donald ... 65,177,251
L80 ... 1,271
labs(value) ... 4,85,95,105,112,154,210,217,232,289,293

Lattice C Compiler ... 175
ldexp(x,n) ... 155
left-justification ... 190
LIB.COM ... 146
link(path1,path2) ... 286
ln(x) ... 156
log(x) ... 17,18,30,31,156,157,234
log10(x) ... 157
logarithm ... 156,157
logged() ... 125,158,218
loginv() ... 159
longjmp(env,v),setjmp(env) ... 58,71,237,238,239
lowmem() ... 139,160
LRU method ... 209,292
lsearch(key,base,n,w,cmp) ... 27,161
ltoa(x,s) ... 163
M80 ... 25,112,226,271
machine-dependent ... 142,183
macros ... 150,151,277
makefcb(fcb,file) ... 38,88,164
malloc(m) ... 29,103,165,215
mantissa ... 155
math.h ... 4,156,167,171
mathpack ... 14,114
mathsort ... 65,66,67,68,177
max(x,y) ... 167
MAXCHN[] ... 37,95,145,236
maxfiles ... 76
maxsector ... 23,32
maxtrack ... 23,32
memsize() ... 139,168,225
mice,mouse accessory ... 142,183
Micropro ... 276
Microsoft ... 1,170,252,263
midstr(str1,str2) ... 169
min(x,y) ... 171
mknod(path,mode,dev) ... 286
mktemp(name) ... 172
modem ... 142,183
modf(x,nptr) ... 173
monitor(lowpc,highpc,buffer,bufsize,nfunc) ... 288
motorola ... 129,136,139,200,202
mount(spec,dir,rwflag) ... 286
moveMEM(dest,source,n) ... 174,254,258
movmem(source,dest,n) ... 175
multiple-argument ... 52,71,109,250
mycmp(p,q) ... 138,203,242
myctrlb() ... 58
mypassword ... 133
n=peekb(0x80) ... 62
Newton Method ... 248
nice(incr) ... 286

no-echo ... 17,30,35,60,118,214
nomatch(n,f,t) ... 76
non-interrupt ... 22,35
non-portable ... 16,100,163,198,209,226,292
non-ripple memory move ... 175
numsort(n,table,scrap) ... 20,27,177
obsolete Unix calls ... 86
open(name,access) ... 48,179
opena(name,access) Unix open ... 179,181
openb(name,access) Binary open ... 181
outport(port,value) ... 142,183
p_fin,p_fout ... 34,105
padding ... 190,193
PASCAL Language ... 85
pause() ... 286
PC-DOS operating system ... 41
PC_BIN,GC_BIN,Cmode ... 21,22
pcTose(fp) ... 286
peekb(addr) ... 184
peekl(addr) ... 66,184
peekw(addr) ... 26,38,67,184
pens, light ... 142,183
perror(s) ... 70,185
pfFLOAT ... 7,36,55,64,73,77,91,107,140,155,173,185,207,228,244,272
pfLONG ... 16,38,66,154,184,187,229
PI ... 7,8,9,10,11,12,13,272,273
pipe(fd) ... 286
plotters ... 142,183
pokeb(addr,x),pokel(addr,x),pokew(addr,x) ... 187
polynomial ... 140
portability ... 151,170,180,182,209,239
pow(x,y) ... 73,188,248
pow10(y) ... 189
precision ... 114,190,191,192,193,228
Prentice-Hall ... 138,242
printf(control,arg1,arg2,...) ... 108,119,121,190,249
prnt_1(),prnt_2 ... 192,193
processforeignfiles() ... 22
profil(buff,bufsize,offset,scale) ... 286
pseudo-random ... 208
ptrace(request,pid,addr,data) ... 286
pump(buffer,file) ... 179,181
pumpword(fp,x) ... 202
push-back ... 283,284
put2048(fp,buf) ... 293
putc(x,fp) ... 52,80,95,101,115,195,276,277,294,295
putcbinary(x,fp) ... 22,197
putchar('\7') ... 45
putchar(x) ... 47,60,81,98,110,199,236
putl(x,fp) ... 200
putpwent(p,fp) ... 286
puts(s) ... 152,153,201,290

putw(n,fp) ... 80,202
qsort(base,n,s,cmp) ... 49,203
quanta ... 95,211,294,295,297
quick(lo,hi,base,cmp) ... 27,205,224,251
quicksort ... 203,205
rad(x) ... 55,207,244,273
Raike ... 204,206
rand(),randl() ... 208
random ... 5,18,31,82,95,263
rdDISK() ... 209
re-direction ... 148,199,253,278,284.295
read (fp,buffer,count) C/80 read ... 211,294
read[fd,buf,n) Unix read ... 28,84,102,212
reada(fd,buffer,n) Unix read ... 102,212
readb(fd,buffer,n) Binary read ... 213
realloc(ptr,s) ... 215
recursive ... 194,247
rename(oldname,newname) ... 18,31,217
report(fp) ... 112
reset() ... 218,234
reverse(str) ... 219
rewind(fp) ... 5,220,221,294
rewind(fp) code macro ... 221
rindex(str,c) ... 222
Ritchie ... 242
robotics ... 71
ROM ... 52,71,233
RST instruction ... 3
run(cmd) ... 223,272,279
Rutter ... 25,204,226,243,285
Sailor ... 51,144
sbinary(x,v,n) ... 27,162,224
sbrk(n) ... 160,225
scanf(control,&arg1,&arg2,...) ... 227,228,230,246,284
scientific ... 191
scrap[10+256] ... 177
secant ... 10
SECSIZ ... 127,128
sectors/track ... 127
seek(fp,offset,type) C/80 standard ... 80,95,145,231,284
seekend(fd) ... 97,99,180,182,233
seldsk(x) ... 127,234
setatt(name,code) ... 43,235
setbuf(fp,buffer) ... 236
setgid(gid) ... 46,287
setgrent() ... 288
setjmp(env) ... 58,71,237,239
SETJMP.H ... 237,239
setmem() ... 85
setpgrp() ... 286
setpwent() ... 288
setuid(uid) ... 46,287

sfFLOAT ... 109,228,230,250,287
sfLONG ... 109,229,230,250
sfree(n) ... 29,95,103,146,147,166,178,240
sgtty ... 286
Shell-Metzner sort ... 251,252
shells(p,n,cmp) ... 27,162,241
SID ... 2,3
Sieve of Eratosthenes ... 85,275
signal() ... 243
sin(x),sine ... 11,70,185,244,245,273
sinh(x) ... 245
SIZBLOCK ... 166
sleep(seconds) ... 288
sob(s) ... 75,93,246
sorted ... 122,124,224,251,252
SPG (sectors per group) ... 124,126,127
sprintf(s,control,arg1,arg2,...) ... 100,114,152,247
SPT (sectors per track) ... 127
sqrt(x) ... 248
srand(seed) ... 208
sscanf(s,control,&arg1,&arg2,...) ... 249,250
sscanf return kludge ... 250
ssort2(size,table,reclen) ... 27,122,125,162,251
ssort3(size,&table) ... 27,162,252
stat(name,buf) ... 287
stdout ... 279
Steele & Harbison ... 29,138,141,190,222
stime(tp) ... 287
strcat(str1,str2) ... 254
strchr(s,c) ... 141,255
strcmp(str1,str2) ... 49,137,138,256
strcpy(str1,str2) ... 90,257
strcspn(s,set) ... 259
strlen(str) ... 141,260
strncat(str1,str2,n) ... 85,261
strncmp(str1,str2,n) ... 262
strncpy null fill warning ... 263
strncpy(str1,str2,n) ... 62,85,263
strpbrk(s,set) ... 264
strpos(s,c) ... 265
strrchr(s,c) ... 222,266
strrpbrk(s,set) ... 267
strrpos(s,c) ... 268
strspn(s,set) ... 269
stub ... 97,243
swab(s,d,n) ... 270
switch(setjmp(env)) ... 71,238
sync() ... 287
system(s) ... 41,271
tablets, graphics ... 142
tangent,tan(x) ... 12,13,272,273,274
tanh(x) ... 274

Taylor polynomial ... 273
time(tloc) ... 287
timer(n) ... 142,169,183,275
times(buffer) ... 287
toascii(x) ... 276
toint(x) ... 277
tokens(table,cmdtail) ... 75,76,143,278
tolower(x) ... 280
Toolworks, The Software ... 94,114,210,280,293
TOT_SZ ... 25,34,226
toupper(x) ... 61,277,281
TPA (transient program area) ... 39,40
transcendental ... 14,70,114,186
ttyname(fd) ... 282
type-aheads ... 22,35,59
typedef ... 239
UART (universal async rec/trans) ... 142,183
uflush() ... 82
umask(cmask) ... 287
umount(spec) ... 287
uname(name) ... 287
unbuff(fd) ... 240
unbuffered ... 180,182
ungetc(x,fp) ... 80,230,283,284
union ... 107,187
UNIXIO.H ... 285
unix I/O and system functions ... 285
unlink(filename) ... 160,175,217,289
ustat(dev,buf),utimbuf,utime(path,times) ... 287
utoa(n,s),utoab(n,s),utoao(n,s),utoax(n,s) ... 290
wait(stat_loc) ... 287
waitz(n) ... 291
Waite and Cody ... 244
wakeports() ... 71
whatdevice(fp) ... 282
while(keystat()) ... 153
Wiley, John & Sons ... 25,204,226,243,285
Wizard C Compiler ... 175
Wordstar (Micropro) ... 276
wrDISK(trk,sec,dsk,buf)) ... 292
write (fp,buffer,count) C/80 standard ... 293,294
write(fd,buffer,n) Unix write ... 48,84,95,179,181,293,294,295,297
writea(fd,buffer,n) Unix write ... 115,295
writeb(fd,buffer,n) Binary write ... 57,196,199,296
write-protect ... 18,31,235
Z100 ... 208,275,291
Z29 ... 52
Z80 CPU ... 142,174,176,178,183,270,291
Z90/H89 Zenith Computer ... 183,209,292

Copyright (1985) by G.B.Gustafson and Viking C Systems
All Rights Reserved

Each separate function is documented according to the following scheme:

- o **Purpose**
- o **Function Header**
- o **Returns**
- o **Example**
- o **Notes**

Function descriptions are listed alphabetically. It is normal to use just one page per function. Use the headline at the top of the page to locate a particular function.

o **Purpose.** A sentence or two which attempts to describe what the library function or variable is supposed to do or accomplish.

o **Function Header.** For each function in the library archive, we extract the function header and explain the nature and purpose of the variables in the argument list, plus the return value of the function.

1. The function name and the order of its arguments.
2. The function return value, e.g., long, int, float, char* or void. Return values are 16 bits or 32 bits.
3. The data storage class of each argument. Each argument uses 16 bits (2 bytes) except for float and long, which use 32 bits (4 bytes). All pointer types use 16 bits.

o **Returns.** The function can return values in several ways, which are documented in this section of the function description:

1. The function returns a value in the processor registers, which is directly usable by language C, e.g., i = f(x).
2. Variables passed in the argument list of the function are pointers to data locations. The function may alter the contents of these data locations.
3. Global variables may be altered by a function, e.g., the error return variable **errno** in the transcendental function library.
4. Disk I/O can change the contents of file buffers, and the position of the file pointer, e.g., getc() and fseek().

o **Example.** Where possible, an example or two is given that illustrates how to use the function. Examples tend to be test programs, e.g., a complete main program that illustrates the central ideas.

o **Notes.** Each function has implementation notes or special problems that are unique to its usage. Included here are cross-references to other functions and the limitations of usage.

The _exit Function

Purpose:

Abort exit. Does not flush buffers nor close the file control block to the disk directory.

Function Header:

```
#include <stdio.h>
VOID _exit()            This is a macro equivalent to a_exit().
                        Stdio.h contains the macro definition.

VOID a_exit()           Actual function name in CLIB.REL.
                        See also CCMAIN.CC.
```

Returns:

Nothing. Exits to warm boot.

Example: Use of _exit() to purge contents of re-direction files.

The command line is: A>main >foo.bar. The file foo.bar will have zero records.

```
#include <stdio.h>
main()
{
     printf("This message will not make it to disk\n");
      _exit();
     printf("This line will never be printed\n");
}
```

Notes:

- Standard abort exit. Calls BDOS function 0, but does not flush buffers or close files.
- Due to problems with the Microsoft linker, the internal symbol used is A_EXIT rather than _EXIT.
- The correct Unix symbol is _exit(). Source code should use this symbol exclusively. The global symbol used by the link editor L80.COM is A_EXIT.

The abort Function

Purpose:

Debugger support for C programs. Can be used with DDT, SID.

Function Header:

```
abort(code)             Abort exit to DDT
int  code;              Code passed to machine register HL
```

Returns:

```
Caller                  If DDT is not loaded
DDT                     If DDT & C program loaded together
```

Example: Call sbrk() for more memory until the request fails, then break to DDT.

```
main(argc,argv) int argc; char **argv;
{
char *sbrk();

        if(sbrk(10240) == (char *)-1) abort(1);
        if(sbrk(10240) == (char *)-1) abort(2);
        if(sbrk(10240) == (char *)-1) abort(3);
        if(sbrk(10240) == (char *)-1) abort(4);
        if(sbrk(10240) == (char *)-1) abort(5);
        if(sbrk(10240) == (char *)-1) abort(6);
        if(sbrk(10240) == (char *)-1) abort(7);
}
```

This program jumps to DDT the first time it fails to obtain another 10k of memory from sbrk(). To re-enter the C program from DDT, note the DDT stop address and do a GO command at that address plus 1 (e.g., -G0a31 if the break was at 0a30 hex).

For example, if the failure occurred at level 5, then DDT would intercept inside the abort() code with HL=5. You can step through the rest of the program, each time re-entering DDT with HL=6, HL=7, until finally the exit() is reached.

WARNING: This abort() is set up for a Heath/Zenith machine using CP/M 2.2 with interrupt 7 reserved for DDT. To get it to function under other CP/M versions or on other hardware, it may be necessary to re-assemble the source code.

Notes:

- The value of the argument is in register HL before the call and in register BC after the call.
- To use DDT, it must be invoked with the program:

  A>DDT MYPROG.COM

- Warning: Like all debugger code, do not leave it in place in the final program.
- This implementation uses the standard Heath CP/M RST location for DDT/SID. It should be a NOP if DDT or SID is not loaded with the C program.
- Both DDT and SID corrupt the interrupt vectors for RST 7. Object code with abort() calls is not end-user object code. Remove abort() calls by placing the source code **abort(){}** with main().
- If your system is different, then it MUST be changed for the hardware and re-compiled.

The abs Function

Purpose:

Computes the absolute value of an integer argument.

Function Header:

```
int abs(x)
int x;                    Integer to transform.
```

Returns:

The positive integer obtained from x by removing the sign, i.e., returns x if x was already positive, else returns -x.

Example: Compute the absolute value of the difference between two numbers a and b.

```
#include <stdio.h>
main()
{
char s[129];
int a,b;

     printf("Enter number a: ");
     gets(s); a = atoi(s);
     printf("Enter number b: ");
     gets(s); b = atoi(s);
     printf("abs((%d) - (%d)) = %d\n",a,b,abs(a-b));
}
```

Notes:

- The abs() function used here is an honest function that works only on integers. It fails on long integers and floats.
- For long integers, use labs().
- For floats, use fabs().
- Most libraries assume the abs() function is a macro defined in the STDIO.H header file or in MATH.H. Such functions definitely have side effects. Beware when you port the code. Engineer against side effects when you write it for the first time.

The access Function

Purpose:

Checks legal access to a file. Under CP/M, the check consists of a search for a matching directory entry, plus matching file attributes ($R/O, $R/W, $DIR, $SYS).

Function Header:

```
int  access(name,mode)        Access boolean 0 or -1
char *name;                   File name string pointer
int  mode;                    Unix mode integer, see below
```

Returns:

```
 0            access allowed (file exists with attributes)
-1            access denied (file not found or bad attributes)
```

Example: Look up the file entered on the command line to see if it is in the disk directory and can be updated as a random file in read/write mode.

```
#include <unix.h>
#include <stdio.h>
help()
{
    puts("Usage: A>main filename"); exit();
}

main(argc,argv) int argc; char **argv;
{
FILE *fp,*fopen();
char buf[128];
int access();
    if(argc <= 1) help();
    if(access(argv[1],02)) puts("File not found or $R/O");
    fp = fopen(argv[1],"a+");
    if(fp == (FILE *)0) { puts("Bad open"); exit();}
    rewind(fp);
    fgets(buf,80,fp);
    fputs(buf,stderr);
}
```

Notes:

- The name is a pointer to a null-terminated file name. If the drive is missing, then it is assumed to be the currently logged drive.

- The mode value is the Unix mode, which under CP/M causes a directory lookup to see if the protection bits match:

UNIX MODE	DESCRIPTION OF ACTION	CP/M ATTRIBUTE
00	check directory access	file exists
01	check exec access	$DIR
02	check write access	$R/W
04	check read access	file exists

- CP/M-80 only checks to see if the file is in the default directory. No directory search path is performed.

The acos Function

Purpose:

Compute the arc cosine of real float value x. The value x is assumed between -1 and 1.

Function Header:

```
float acos(x)            Arc Cosine.
float x;                 Float value for computation.
```

Returns:

```
   0       Range error, set errno = EDOM
 angle     No error, answer in radians 0..PI
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv)  int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float acos();
     errno = 0;
     printf("Enter a float: "); gets(s); x = atof(s);
     printf("acos(%f) = %f, errno = %d\n",x,acos(x),errno);
}
```

Notes:

o The method used is acos(x) = PI/2 - asin(x).

The acot Function

Purpose:

Compute the arc cotangent of real float value x. The value x is assumed between -INF and +INF.

Function Header:

```
float acot(x)           Arc Cotangent.
float x;                Float value for computation.
```

Returns:

angle answer in radians 0 .. PI approximately.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float acot();
     errno = 0;
     printf("Enter a float: "); gets(s); x = atof(s);
     printf("acot(%f) = %f, errno = %d\n",x,acot(x),errno);
}
```

Notes:

o The method used is acot(x) = PI/2 + atan(-x).

The acsc Function

Purpose:

Compute the arc cosecant of real float value x. The value x is assumed between -INF and +INF.

Function Header:

```
float acsc(x)           Arc Cosecant.
float x;                Float value for computation.
```

Returns:

angle Answer in radians -PI/2 .. PI/2 approx

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float acsc();

     errno = 0;
     printf("Enter a float: "); gets(s); x = atof(s);
     printf("acsc(%f) = %f, errno = %d\n",x,acsc(x),errno);
}
```

Notes:

- The method used is acsc(x) = asec(x/sqrt(x*x-1)).

The asec Function

Purpose:

Compute the arc secant of real float value x. The value x is assumed between -INF and +INF.

Function Header:

```
float asec(x)           Arc Secant.
float x;                Float value for computation.
```

Returns:

```
0         Range error, set errno = EDOM.
angle     No error, answer in radians 0 .. PI.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float asec();
     errno = 0;
     printf("Enter a float: "); gets(s); x = atof(s);
     printf("asec(%f) = %f, errno = %d\n",x,asec(x),errno);
}
```

Notes:

o The method used is:

```
asec(x) = atan(sqrt(x*x - 1)) - (x >= 0 ? 0 : PI);
```

The asin Function

Purpose:

Compute the arc sine of real float value x. The value x is assumed between -1 and 1.

Function Header:

```
float asin(x)           Arc Sine.
float x;                Float value for computation.
```

Returns:

```
0         Range error, set errno = EDOM.
angle     No error, answer in radians -PI/2 .. PI/2.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv)  int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float asin();
     errno = 0;
     printf("Enter a float: "); gets(s); x = atof(s);
     printf("asin(%f) = %f, errno = %d\n",x,asin(x),errno);
}
```

Notes:

o The method used is asin(x) = atan(x/sqrt(1 - x*x)).

The atan Function

Purpose:

Compute the arc tangent of real float value x. The value x is assumed between -INF and +INF.

Function Header:

```
float atan(x)
float x;            Float value in range -INF to +INF.
```

Returns:

```
angle     Answer in radians -PI/2 .. PI/2 approx
0.0       ERANGE error
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float atan();
     errno = 0;
     printf("Enter a float: "); gets(s); x = atof(s);
     printf("atan(%f) = %f, errno = %d\n",x,atan(x),errno);
}
```

Notes:

- The method used is rational approximation, Cheney & Hart.
- Constants from C&H, 6.5.21, 6.5.22, Table 5097.

The atan2 Function

Purpose:

Compute the arc tangent of real float value y/x to find the angle in radians between the x-axis and the ray through (0,0) and (x,y).

Function Header:

```
float atan2(x,y)
float x,y;              Float values for the point (x,y).
```

Returns:

```
0             Error x=y=0, errno = EDOM
angle         Answer in radians -PI .. PI approx
PI/2          x=0, y>0
-PI/2         x=0, y<0
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,y;
extern int errno;
float atof();
float atan2();
  errno = 0;
  printf("Enter float x: "); gets(s); x = atof(s);
  printf("Enter float y: "); gets(s); y = atof(s);
  printf("atan2(%f,%f) = %f, errno = %d\n",x,y,atan2(x,y),errno);
}
```

Notes:

- The function exists to handle the likely boundary cases. Users should study the return values carefully and include error checks in all source code that uses atan2().

The atof Function

Purpose:

Converts a null-terminated string of decimal digits into an internal format 32-bit floating point number.

Function Header:

```
float atof(s)
char *s;                    String of decimal digits.
```

Returns:

Floating point number, single-precision, 32-bits.

Example: Read a float from the console, print its value.

```
#define pfFLOAT 1
#include <stdio.h>
main()
{
float f,atof();
char s[129];

     printf("Enter a float: ");
     gets(s);
     f = atof(s);
     printf("float f = %f\n",f);
}
```

Notes:

- The cast of atof() is (float). It must be declared before use. Failure to do so may crash your machine.
- Floats are 32 bits (4 bytes). Storage of a float is documented in the C/80 Mathpack. See also the Transcendental function library for information about the floating point exponent.
- The sources for ATOF and FTOA are part of the MathPack. Use of atof() invokes the float library.
- Float atof() is used to decode strings in fixed or scientific formats. Strings like "84745475.237237" or "1.0e-16" are acceptable. Atof() is a Unix function.
- Float allows decimal points, leading +-, exponent e or E, and a sign in the exponent. A floating point number cannot contain imbedded white space. Non-digits cause the decoding to stop. This includes white space or the end of the string.

The atoi Function

Purpose:

Converts a null-terminated string of decimal digits into a 16-bit internal format integer.

Function Header:

```
int atoi(s)
char *s;                        String of decimal digits.
```

Returns:

Integer in the range -32767 to 32767. The result is signed.

Example: Read an integer from the console, print its value.

```
#include <stdio.h>
main()
{
int i;
char s[129];

     printf("Enter a signed integer: ");
     gets(s);
     i = atoi(s);
     printf("i = %d\n",i);
}
```

Notes:

- The string may have leading white space and a leading sign next to the first decimal digit. Digits must be from the character set "0123456789". No hexadecimal digits are allowed. Leading zeros do not signify octal for this function - they are ignored.
- Unsigned integers greater than 32767 will be treated as long integer data by the compiler, unless a suitable cast is imposed on the number.
- The internal storage of an integer is two bytes (16 bits), stored in the usual 8080 reverse format. Integers typed at the terminal are in ascii decimal format, using the character set '0',...,'9'.
- Non-digits cause the decoding to stop. This includes white space or the end of the string.

The atol Function

Purpose:

Converts a null-terminated string of decimal digits into an internal format 32-bit signed long integer.

Function Header:

```
long atol(s)
char *s;                        String of decimal digits.
```

Returns:

Signed long integer.

Example: Read a long integer from the console, print its value.

```
#define pfLONG 1
#include <stdio.h>
main()
{
long i,atol();
char s[129];

     printf("Enter a signed integer: ");
     gets(s);
     i = atol(s);
     printf("long i = %ld\n",i);
}
```

Notes:

- The cast of atol() is (long int). It must be declared before use. Failure to do so may crash your machine.
- The string may have leading white space and a leading sign next to the first decimal digit. Digits must be from the character set "0123456789". No hexadecimal digits are allowed. Leading zeros do not signify octal for this function - they are ignored.
- Unsigned integers greater than 32767 will be treated as long integer data by the compiler, unless a suitable cast is used.
- Long integers are 32 bits (4 bytes). Storage of a long integer is such that you may address its lower order 16 bits as an integer, without error. However, this is highly non-portable (68000 CPU, for example).
- Non-digits cause the decoding to stop. This includes white space or the end of the string.
- Use of atol() invokes the long integer library.

The bdos Function

Purpose:

Provides execution of basic BDOS functions as described in the Digital Research CP/M 2.2 Interface Guide.

Function Header:

```
int bdos(code,DE)
int code;               Function code 0 to 37, 40 (see below)
char *DE;               Parameter, 0 if not used. See below.
```

Returns:

Value or error return for the BDOS function. Returns the value in the usual C/80 interface register, which is HL.

Example:

```
bdos(11,0);     console status
bdos(1,0);      console input
bdos(2,'A');    print 'A' to the console
bdos(9,"A$");   print string "A" to console
bdos(26,p);     set DMA address to p
bdos(14,1);     log in drive A:
bdos(6,255);    raw console input no-echo
bdos(6,x);      raw character output, char = x
```

WARNING: All arguments must be used. Bdos() requires two arguments of 16-bits each. This interface is not portable. For example, BDS-C does it differently and DRI CP/M-68K uses a LONG 32-bit argument in place of DE or BC. Return values across systems are not consistent. This function is a compromise due to the lack of a standard.

Notes:

BDOS FUNCTION NUMBERS

```
0    system reset
1    console input
2    console output, DE=data
3    reader input
4    punch output, DE=data
5    list output, DE=data
6    direct console, DE=255 for input, else output
7    get I/O byte
8    set I/O byte, DE=I/O byte
9    print string terminated with '$', DE=string address
10   read console buffer (formatted)
11   get console status ( -1 if ready)
12   get version number of CP/M
13   reset disk system
14   select disk drive and log in disk, DE=drive number
15   open existing file, DE=FCB
16   close FCB to disk directory
17   search for first, DE = FCB
18   search for next
19   delete file, DE=FCB
20   read sequential, DE=FCB
21   write sequential, DE=FCB
22   create file, DE=FCB
23   rename file, DE=FCB in special format
24   get login vector, returns HL in special format
25   get current disk, range 0 to 15
26   set DMA address, DE=address for disk operations
27   get address of alloc vector, returns HL
28   write-protect drive
29   get read-only vector, returns HL
30   set file attributes, DE=FCB
31   get address of disk parameter block (DPB), returns HL
32   set or get user code, DE=255 to get, else set code in DE
33   read random, DE=FCB in 36-byte format
34   write random, DE=FCB in 36-byte format
35   compute file size, DE=FCB
36   set random record, DE=FCB in 36-byte format
37   reset drive, DE=drive vector
40   write random with zero fill, DE=FCB
```

o Bdos() jumps to the library routine ccBDOS(). The latter allows 1, 2 or 3 arguments. They in turn are not portable, but present a portable interface across machines.

The binary Function

Purpose:

Binary search of a sorted integer array for an integer key value.

Function Header:

```
int binary(x,v,n)
int x;                  Key value to find in the array v[].
int v[];                Sorted integer array for lookup.
int n;                  Dimension of the array v[].
```

Returns:

```
-1          Failure. The test (v[k] == x) failed for 0 <= k < n.
 k          Success. A value of TRUE was found for (v[k] == x).
```

Example: Look up a terminal type in a table.

```
static char *term[] = {
     "VT52","ADDS","TELEVIDEO","Z29","H19","VISUAL200",
     "VISUAL500","WYSE","TELERAY","QUME"
};
static int v[] = {
     5,7,8,9,10,15,16,18,21,25     /* WARNING: MUST BE SORTED! */
};
#include <stdio.h>
main()
{
int k,x;
char s[129];
     printf("Possible codes: 5,7,8,9,10,15,16,18,21,25\n");
     printf("Enter terminal code: ");
     gets(s);
     x = atoi(s);
     k = binary(x,v,10);
     if(k == -1) puts("Not found");
     else puts(term[k]);
}
```

Example: Print an error message from its error code.

```
static
int v[] = {
  4,12,37,65,101,107,110,201,203,209
};
static
char *err[] = {
     "No disk space","File name invalid","Printer not ready",
     "Invalid character","No disk in drive","Digit expected",
     "Decimal point expected","Exponent expected","Bad drive name",
     "Directory full"
};
#include <stdio.h>
prerror(x)
int x;
{
int k;
     k = binary(x,v,10);
     if(k == -1) puts("Undefined error");
     else puts(err[k]);
}
```

Notes:

- The assumption that the integer array v[] is SORTED is essential. It will probably hang the machine if the array is not sorted.
- A normal application uses sorted data entered in the program by hand as initializers for an integer array v[].
- To obtain a sorted integer array on the fly, use numsort().

The Binary Flags for C/80

Purpose:

Sets I/O mode as ASCII TEXT or BINARY. Files are set by PC_BIN and GC_BIN, while the console is set by Cmode.

Function Header:

```
extern char Cmode;          Character mode echo = 0
                            Line mode echo = 1
                            Character mode no echo = 2
extern char PC_BIN;         ASCII mode = 0
                            BINARY mode = 1
extern char GC_BIN;         ASCII mode = 0
                            BINARY mode = 1
```

Returns:

Binary mode eliminates all translation. This mode almost simulates UNIX, except for the identity of the newline character.

Ascii text mode throws away carriage return (0x0D) on input and on output translates linefeed to carriage return plus linefeed. In addition, end of file is detected from ctrl-Z. These modes are independent of the open mode, so fill characters at end of file are unchanged from those set at open.

The variable Cmode controls input mode only for the console.

Echo mode prints the character at the time of input. Ascii mode causes echo of carriage return and linefeed for a carriage return only. No echo mode allows input without console echo.

Console Line Mode requires a carriage return to end the input. During input, all CP/M editing features and IObyte functions are in force. Line Mode always echoes.

Console Character Mode is binary input mode for the console. The character is available immediately. Character mode may either echo (Cmode = 0) or not echo (Cmode = 2).

Example: Manually set all I/O as binary in order to process foreign data files on a CP/M machine. Then reset the modes for normal processing.

```
#include <stdio.h>
main()
{
extern char PC_BIN,GC_BIN,Cmode;

     PC_BIN = GC_BIN = 1; Cmode = 2;
     ProcessForeIgnFiles();
     PC_BIN = GC_BIN = 0; Cmode = 1;
     printf("Operation completed\n");
}
```

Notes:

- o These features are not present under UNIX nor are they to be expected under most systems. However, simulation of the features is usually possible by writing assembly language interface routines on the host.

- o The function getbyte() uses a mode change Cmode = 2 to get the character. But the value of Cmode is restored after the call.

- o The functions getcbinary() and putcbinary() set and reset the global modes PC_BIN, GC_BIN on each call. They may also set and reset Cmode.

- o Console input is buffered to 136 characters under C/80. There is special software in place which loops after each console input in order to pick up type-aheads and function keys. The number of loops is controlled by a variable

  ```
  extern char CCtlCk;
  ```

  This variable is currently set at 6, which seems to work well with 3-character function keys and slow auto-repeat. It may not work very well with non-interrupt driven keyboards. Your custom C code may set it to a higher value, if needed.

The bios Function

Purpose:

Provides execution of basic BIOS functions as described in the Digital Research CP/M 2.2 Interface Guide.

Function Header:

```
int bios(code,BC)
int code;              Function code 1 to 15, see below.
char *BC;              Parameter, 0 if not used. See below.
```

Returns:

Value or error returned by the BIOS function, in the normal C/80 interface register, which is HL.

Example:

```
x = bdos(2,0);          console status
x = 255 & bios(3,0);    console input
```

WARNING: All arguments must be used. Bios() requires two arguments of 16-bits each. This interface is not portable. For example, BDS-C does it differently and DRI CP/M-68K uses a LONG 32-bit argument in place of DE or BC. Return values across systems are not consistent.

BIOS FUNCTION NUMBERS

1 warm boot
2 console status, -1 if ready
3 console input, raw
4 console output, raw, BC = char
5 list output, raw, BC = char
6 punch (AXO:) output, raw, BC = char
7 reader (AXI:) input, raw
8 home disk, BC = drive number
9 select disk drive, BC = drive number 0 to 15
10 set track, BC = track number 0 to maxtrack
11 set sector, BC = sector (usually 1 to maxsector)
12 set DMA address, BC = DMA address
13 read sector
14 write sector, BC = 0 (normal), 1 (directory)
15 list status, returns -1 if ready
16 sector translate, BC=logical sector, DE=translate table addr
Must use 3-argument ccBIOS().

o Bios() jumps to the library routine ccBIOS(). The latter allows 1, 2 or 3 arguments - ccBIOS() is not portable, but presents a portable interface across machines.

The brk Function

Purpose:

Sets the upper bound of the program to an absolute address.

Function Header:

```
int  brk(address)          Integer return 0 or -1
char *address;             Address to set as new
                           program break address.
```

Returns:

```
 0        on success
-1        on failure
```

Example: Reset a C program on exit to its pristine state so that sbrk() sees the same memory on each re-start.

```
extern char *end,*etext,*edata;
brk(end);
```

Example: Undo a sequence of calls to sbrk() that have used n bytes of memory, without knowing the base address.

```
unsigned n;
char *sbrk();
brk(sbrk(0)-n);
```

Notes:

- brk() checks for text, data and stack over-write.

- The symbols end, etext, edata have these meanings:

end	The next usable address after program load.
etext	The physical end of the program text before the file buffers and FCB buffers begin.
edata	Same as etext for C/80 because code and data are in the same place.

- The externals end, etext, edata are defined in the assembly module ETEXT.C as follows:

```
end:   DW  $END##+TOT_SZ##
edata: DW  $END##
etext: DW  $END##
```

- The double-# marks a symbol as external for M80, e.g, $END## is equivalent to EXTRN $END.

- The object code uses addresses 100h to $END, that is, the length of the compiled program is (etext-0x100).

- TOT_SZ is the total size of the file buffer and file control block (FCB) area that immediately follows the object code. See STDIO.H. The base address of the area is end (= $END). The area uses TOT_SZ bytes with last address (etext - 1).

- Reference: For end, edata, etext, see Banahan & Rutter, The Unix Book, Wiley Press (1984), p 198. For a description of brk() and sbrk(), see Banahan, p 205.

The bsearch Function

Purpose:

Binary search of an arbitrary **sorted** table of info. Requires a special compare function to match the internal table structure.

Function Header:

```
char *bsearch(key,base,n,w,cmp)
char *key;                      Key for table search.
char *base;                     Base address of sorted table.
int n;                          Table size in elements.
int w;                          Width of each table element.
int (*cmp)();                   Compare function cmp(key,tbl)

int cmp(key,tbl)                User-supplied compare routine.
char *key;                      key = address of the key
char *tbl;                      tbl = address of table entry;
```

Returns:

```
(char *)        Location of table match
(char *)0       Not found
```

Example: Search a sorted list of strings for a key match.

```
#include <unix.h>
#include <stdio.h>
static char *tbl[] = {
  "ABC","AB","EFG","HJK","M","abc","abd","bcf"
};

cmp(p,q) char *p,*q;
{
  return strcmp(p,peekw(q));
}
main()
{
char key[129];
char *p;
char *bsearch();
  printf("Enter a 1 to 3 character string: ");
  gets(key);
  p = bsearch(key,tbl,8,2,cmp);
  if(p == (char *)0) puts("Not found");
  else puts(peekw(p));
}
```

Note in particular the table width is sizeof(char *). The strings are not all of the same width. The table tbl[] is a table of pointers, which causes the unusual addressing modes seen in the example. We used the special function peekw() instead of **-pointers to document ideas.

Notes:

- This is a UNIX function. See also lsearch(), binary(), sbinary(), ssort2(), ssort3(), numsort(), dsort(), dsort16(), shells(), qsort(), quick(), heap().
- The addressing modes required for string handling are unnatural. See the example above to get it right.
- The complete source follows, since this is an often used but poorly understood function.

```
char *bsearch(key,base,n,w,cmp)
char *key;
char *base;
int  n;
int  w;
int  (*cmp)();
{
static
char    *q,*p;
static
int     i,j;
  q = base;
  while(n > 0) {
    i = n >> 1;
    if((j = (*cmp)(key,p = q + i*w)) == 0) return p;
    else if(j < 0) n = i;
    else {
      q = p + w; n = n - i - 1;
    }
  }
  return ((char *)0);
}
```

The calloc Function

Purpose:

Allocates contiguous memory located between the end of the program and the stack, then initializes it to zero.

Function Header:

```
char *calloc(m,s)              The address of the contiguous area
unsigned m,                    The number of elements
         s;                    The size in bytes of each element
```

Returns:

The base address of the area, on success.
0 if the request fails.

Example: Get buffer space for a block read of a data file, then free the space. The file to be read is entered on the command line as:

```
A>main <datafile
```

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i,n,fd; char *buf;
char *calloc();
  buf = calloc(1024,1);
  fd = fileno(stdin);
  if(buf == (char *)0 || fd == 0) exit();
  n = read(fd,buf,1024);
  for(i=0;i<n;++i) putchar(buf[i]);
  free(buf);
}
```

Notes:

- The number m of elements of size s must be such that m*s does not overflow as an unsigned integer.
- The address returned is the base address of a contiguous area of memory whose byte length is m*s, or 0 for failure.
- Uses malloc() to get raw system memory.
- The maximum request is 65531 bytes under CP/M-80.
- Use free() to release a block. Same as cfree() in the reference manual by Harbison and Steele.
- Two successive calls to calloc() will not in general result in a single block of contiguous memory.
- Calls to calloc() cause a header to be written at the beginning of the block. Malloc() uses the header in an essential way - don't write over it!
- Sfree() and brk() don't know about calloc().

The ccBDOS Function

Purpose:

Provides execution of basic BDOS functions as described in the Digital Research CP/M 2.2 Interface Guide. The naming conventions attempt to be non-conflicting.

Function Header:

```
int ccBDOS(DE,BC)
char *DE;               Parameter. May be omitted if not used.
char *BC;               CP/M BDOS function call number.
```

Returns:

Value or error return for the BDOS function

Bdos calls 12, 24, 27, 29, 31 return an HL vector.

Examples:

```
(int)ccBDOS(11);        console status
(int)ccBDOS(1);         console input
(int)ccBDOS('A',2);     print 'A' to the console
(int)ccBDOS("A$",9);    print string "A" to console
(int)ccBDOS(p,26);      set DMA address to p
(int)ccBDOS(1,14);      log in drive A:
(int)ccBDOS(255,6);     raw console input no-echo
(int)ccBDOS(x,6);       raw character output, char = x
```

Notes:

BDOS FUNCTION NUMBERS

```
0    system reset
1    console input
2    console output, DE=data
3    reader input
4    punch output, DE=data
5    list output, DE=data
6    direct console, DE=255 for input, else output
7    get I/O byte
8    set I/O byte, DE=I/O byte
9    print string terminated with '$', DE=string address
10   read console buffer (formatted)
11   get console status ( -1 if ready)
12   get version number of CP/M
13   reset disk system
14   select disk drive and log in disk, DE=drive number
15   open existing file, DE=FCB
16   close FCB to disk directory
17   search for first, DE = FCB
18   search for next
19   delete file, DE=FCB
20   read sequential, DE=FCB
21   write sequential, DE=FCB
22   create file, DE=FCB
23   rename file, DE=FCB in special format
24   get login vector, returns HL in special format
25   get current disk, range 0 to 15
26   set DMA address, DE=address for disk operations
27   get address of alloc vector, returns HL
28   write-protect drive
29   get read-only vector, returns HL
30   set file attributes, DE=FCB
31   get address of disk parameter block (DPB), returns HL
32   set or get user code, DE=255 to get, else set code in DE
33   read random, DE=FCB in 36-byte format
34   write random, DE=FCB in 36-byte format
35   compute file size, DE=FCB
36   set random record, DE=FCB in 36-byte format
37   reset drive, DE=drive vector
40   write random with zero fill, DE=FCB
```

The ccBIOS Function

Purpose:

Provides execution of basic BIOS functions as described in the Digital Research CP/M 2.2 Interface Guide. The naming conventions attempt to be non-conflicting.

Function Header:

```
int ccBIOS(DE,BC,f)
char *DE;               Parameter. May be omitted.
char *BC;               Parameter. May be omitted.
int f;                  Function code 1 to 15. Required.
```

Returns:

Value or error returned by the CP/M 2.2 BIOS function

Bios calls may return a vector in HL or a byte value.
The C version always returns in HL.

Example:

```
ccBIOS(2);          console status
ccBIOS(3);          console input
ccBIOS('A',4);      print 'A' to the console
ccBIOS(13);         read logical sector of 128 bytes.
```

Notes:

BIOS FUNCTION NUMBERS

```
1    warm boot
2    console status, -1 if ready
3    console input, raw
4    console output, raw, BC = char
5    list output, raw, BC = char
6    punch (AXO:) output, raw, BC = char
7    reader (AXI:) input, raw
8    home disk, BC = drive number
9    select disk drive, BC = drive number 0 to 15
10   set track, BC = track number 0 to maxtrack
11   set sector, BC = sector (usually 1 to maxsector)
12   set DMA address, BC = DMA address
13   read sector
14   write sector, BC = 0 (normal), 1 (directory)
15   list status, returns -1 if ready
16   sector translate, BC=logical sector, DE=translate table addr
     Must use ccBIOS().
```

The ccmain Function

Purpose:

Standard main program for a compiled C program. Performs setup of the console buffer, command line, stack, heap, re-direction files and passes control to routine main(). Normal return to this function drops through exit().

Function Header:

NONE This is not a user-callable function.
It is used implicitly. See the example below.

Returns:

NOTHING ccmain() returns to warm boot.

Example: Write a main program in C.

```
#include <stdio.h>
main()
{
     puts("This is the main program, always called main().");
     puts("ccmain is the startup code for main().");
     puts("The end brace of main() causes return to ccmain().");
     puts("Upon return, ccmain() calls exit().");
}
```

Notes:

- The following functions are called by ccmain():

Copen	< & > redirection file opener, see COPEN.CC
Cexit	buffer emptying and close files, see CEXIT.CC
tokens	command line processor, see TOKENS.CC
C_mcln	move cmd line to Cbuf console buffer, see CMCLN.CC
main	main program, user external

- The routine uses the following global constants and symbols:

EX_SPC	amount of extra space below BDOS, see stdio.h
TOT_SZ	size of initial fcb's & file buffers, see stdio.h
$END	physical end of program, see END.CC

- The entry points provided in ccmain():

VOID exit()	standard exit
VOID a_exit()	abort exit
VOID Ctlb ()	^B jump, language C rules
char *CC_CCP	stack pointer for caller of CCMAIN
int fin	redirection standard input descriptor
int fout	redirection standard output descriptor
char *p_fin	redirection input filename pointer
char *p_fout	redirection output filename pointer
char *Cbuf	console buffer pointer
char Cmode	char mode echo=0, line mode=1, char mode noecho=2
char *CtlB	^B vector address data - extern *char CtlB;

- The end address of the program is saved in the integer variable $LM, accessible only through assembly language. This variable is used by sbrk() to manage the heap. The linkage of ccmain() to your program is controlled in STDIO.H by declaring this variable as EXTERN.

The CCtlCk Variable

Purpose:

Contains the number of loops to be made after console input for the purpose of buffering function keys and type-aheads. The default number of additional loops is 6. Adjustment of the variable contents may be needed for slow terminals and non-interrupt CP/M 2.2 console.

Function Header:

```
extern char CCtlCk;
```

Returns:

After a console input, the keyboard is checked for status and ctrl-C and ctrl-B interrupts. This variable controls the number of loops to be made. If a character is available, then it is buffered into the 136-character console buffer and echoed according to the currently set Cmode flag (0,1 echo but 2 does not).

Example: Set up loop cycles of 12 for a slow terminal and CPU which has no interrupt-driven keyboard. This program is designed to capture function keys and display the constituent characters.

```
#include <stdio.h>
main()
{
extern char CCtlCk;
int x;
  CCtlCk = 12;
  puts("Function key test. Press ctrl-Z to exit");
  while(( x = getbyte() ) != 26) {
     CtlCk();
     printf("%d Dec, %x Hex, %o Oct, %b Bin\n",x,x,x,x);
  }
}
```

Notes:

- The internal operation of CtlCk() is to call the keyboard status routine to see if the user typed a character. If so, then the character is read no-echo and added to the console buffer. If line mode is active, then ctrl-C, ctrl-B will signal exits to the operating system.

- In character mode, a call to CtlCk() is the same as adding available characters to the console buffer. No special action is taken for ctrl-C or ctrl-B.

The ceil Function

Purpose:

Compute float ceiling, the smallest integer greater than or equal to the given number. For example, ceil(-1.1) = -1, ceil(1.1) = 2.

Function Header:

```
float ceil(f)
float f;            Float argument.
```

Returns:

```
g               Where g is a whole number, signed,
                with f <= g and g-1 < f.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv)  int argc; char **argv;
{
char s[129];
float f;
float atof();
float ceil();
     printf("Enter float f: "); gets(s); f = atof(s);
     printf("ceil(%f) = %f\n",f,ceil(f));
}
```

Notes:

- No error codes returned. No need to code for errno.
- Ceil() does not truncate. For x > 0, add 1 and truncate, but for x < 0 truncate.
- No overflow check.
- No auto conversion to FLOAT.

The Cexit_ Function

Purpose:

Flushes buffers and closes the file control block to the disk directory for stream stdout.

Function Header:

```
VOID Cexit_()
```

Returns:

Nothing.

Example: Making a new Cexit_() to simulate systems that flush and close all files on exit.

```
Cexit_()
{
extern char IOch[];
extern int MAXCHN[];
int i,fd;

     for(i=0;i<MAXCHN;++i) {
          if((fd = IOch[i]) != -1) fclose(fd);
     }
}
```

Notes:

- The provided Cexit_() routine flushes and closes stream stdout. But any other open streams will be ignored. Open files will leave orphan disk directory entries with 0 records.

The cfs Function

Purpose:

Compute the virtual file size of a CP/M disk file.

Function Header:

```
unsigned cfs(file)
char *file;              Normal CP/M file name. May not be a
                         device name.
```

Returns:

The file size in 128-byte records. This may not be the right answer if the file was fragmented during creation or if the file is currently open.

Example: Find out the length in bytes of a binary file.

```
#define pfLONG 1
#include <stdio.h>
main()
{
long n;
unsigned x,cfs();
char s[129];
     printf("Enter a file name: ");
     gets(s);
     x = cfs(s);
     n = x*128L;
     if(x != -1)
       printf("%s has length %ld bytes\n",s,n);
     else
       printf("File %s not found\n",s);
}
```

Notes:

o Here is the source code for cfs().

```
unsigned int cfs(file)
char *file;
{
auto char fcb[36];
char *findFIRST();
  makeFCB(fcb,file);
  if(findFIRST('\0',file) == (char *)0) return -1;
  ccBDOS(fcb,35);
  return peekw(fcb+33);
}
```

The chain Function

Purpose:

Loads a CP/M file of arbitrary contents at a given address and then jumps to that address. The file control block area used for the call must be 100 bytes in length and not overlap the space to be used by the newly loaded program. Not a beginners function.

Function Header:

```
VOID chain(tpa,program)
char *tpa                   Load address for program.
char *fcb;                  File Control Block, 100 bytes long.
                            Safe area for the FCB and program
                            loader.
```

WARNING: The area at fcb must be 100 bytes in length and remain protected during the load of the program. You must not chain to a program that can over-write the controlling file control block and the program loader area.

Returns:

Nothing. The routine chain() returns to the caller only in case the file is not found. Otherwise, the COM file is loaded and run. There is no default buffer processing.

Example: Load and run a program that resides above the BIOS. It is assumed that addresses 0xE000 to 0xFFFF are unoccupied and that the file DRIVER.PRE has been assembled to load at 0xE100. The code at 0xE100 should do its thing and exit to warm boot.

```
#include <stdio.h>
main()
{
     makeFCB(0xE000,"DRIVER.PRE");
     chain(0xE100,0xE000);
     puts("DRIVER.PRE not found");
}
```

Notes:

- The routine makeFCB() makes a CP/M standard file control block of 36 bytes from a string that is supposed to represent a file name.
- The fcb area is an array fcb[100]. The first 36 bytes store the file control block for the file to be loaded. In the remaining 64 bytes is stored a copy of a relocatable loader program which reads the file off the disk into memory at the set TPA address. Clearly the area fcb[100] may not intersect the buffer area for the file read.
- The action taken by chain() is to copy the file to the desired tpa address and then jump to the base in order to run the program.
- There is no internal error-checking to see if the programmer is about to crash the system. Indeed, this routine is not for the faint of heart, for it will certainly sustain a few crashes during the debugging stages.
- The ability to load a program anywhere in memory and run it might be seen as a kind of concurrency. It can be thought of as a way to initialize and load device drivers.
- A convenient and safe place to select fcb[100] is the console buffer for C/80 programs. It is in high memory just under FDOS and 136 bytes in length. If the file is missing fails, then no harm comes to the program currently in memory.

The chdir Function

Purpose:

Change the working directory to a new drive and new user area.

Function Header:

```
chdir(s)
char *s;        String of format "A0:" for drive A: user 0.
                User areas are 0 to 15 and drives are A: to P:.
```

The drive is not changed unless it is present. The user area is not changed unless present.

Returns:

Nothing useful. If the drive does not mount, then this function might drop the user out to CP/M.

Example: Change the default drive to B: and the user area to 1.

```
#include <unix.h>
#include <stdio.h>
main()
{
  chdir("B1");
  puts("User 1, default drive B:");
  system("DIR");
}
```

Example: Change to user 15 on the currently logged disk.

```
#include <unix.h>
#include <stdio.h>
main()
{
  chdir("15");
  puts("User 15");
  system("DIR");
}
```

Notes:

o This is for CP/M 2.2 only. Note the analog under PC-DOS.

The chmod Function

Purpose:

Change the protection of an existing file. The CP/M file attributes that can be updated are $R/W, $R/O, $DIR, $SYS.

Function Header:

```
int chmod(name,mode)
char *name;             Name of the file to access
int  mode;              Unix mode number, see below.
```

The effect of chmod() is to set the file flags of a CP/M file as follows:

CP/M MODE	UNIX MODE	UNIX I-NODE PATTERN
$R/W & $DIR	04777	-rwxrwxrwx
$R/O & $DIR	04555	-r-xr-xr-x
$R/O & $SYS	04444	-r--r--r--
$R/W & $SYS	04666	-rw-rw-rw-

Returns:

```
 0      File exists and operation was successful.
-1      File not found, or directory update failed.
        [Write-protect tabs can cause failure]
```

Example: Look up the disk file "A:NAMES", set the file to $R/W and $DIR, open the file for append and add the name "Ritchie", close the file and set its file attributes to $R/O + $DIR.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
FILE *fp,*fopen();
  if(chmod("A:NAMES",04777) == 0) {
    fp = fopen("A:NAMES","a+");
    if(fp == (FILE *)0) { puts("File error"); exit();}
    fputs("Ritchie",fp);
    fclose(fp);
    chmod("A:NAMES",04555);
  }
  else {
    puts("File not found or disk write-protected");
  }
}
```

Notes:

- CP/M considers a file to be:
  readable - exists in the disk directory
  writable - exists and has $R/W attribute
  executable - exists and has $DIR attribute

- We map CP/M flags to permissions as follows:

$DIR	execute permission on
$SYS	execute permission off
$R/W	write permission on
$R/O	write permission off

- Chmod() makes Unix source code compile without errors, even though it may not run properly.

- The octal Unix modes are:

```
04000 Set user id on execution (suid)
02000 Set group id on execution (sgid)
01000 Save text in swap area after execution
00400 owner read permission
00200 owner write permission
00100 owner execute permission
00040 group read permission
00020 group write permission
00010 group execute permission
00004 all others read permission
00002 all others write permission
00001 all others execute permission
```

- For example, 04755 means to set user id, all modes for the user but write permission is turned off for group and all others.

- To turn off all action of chmod(), insert this code segment into code that accesses chmod():

```
setatt()
{
   return 0;
}
```

The CHmode Function

Purpose:

Switches between raw character mode and line mode with built-in editing functions.

Function Header:

```
VOID CHmode(n)
int n;                  Character mode with echo for n = 0.
                        Line mode for n = 1.
                        Character mode no echo for n = 2.
```

Returns:

Nothing useful.

Example: Get a character from the user with echo but no wait for a carriage return and linefeed.

```
#include <stdio.h>
main()
{
     printf("Yes or No? <Y/N>: ");
     CHmode(0);
     printf(toupper(getchar()) == 'Y' ? "\bYES\n" : "\bNO\n");
     CHmode(1);
}
```

Example: Get a character from the user with no echo. No wait for a carriage return and linefeed.

```
#include <stdio.h>
main()
{
     printf("Yes or No? <Y/N>: ");
     CHmode(2);
     printf(toupper(getchar()) == 'Y' ? "YES\n" : "NO\n");
     CHmode(1);
}
```

Notes:

- Uses external global Cmode = 0, 1, or 2 in the switching.

- The backspace method used above in the first example presumes that the terminal can backspace. It does not handle CR/LF answers. The second method works without terminal problems.

- A good tool for handling user options is:

```
CHmode(2);
while(index("YN",toupper(getchar())) == (char *)0)
     putchar('\7');       /* blow horn */
CHmode(1);
```

  This method has the advantage of filtering out all bad input and therefore reducing program bugs as the code is revised.

- A shorter version of the above code is

```
while(index("YN",toupper(getbyte())) == (char *)0)
     putchar('\7');       /* blow horn */
```

- The change to character mode from line mode has its problems. Some CP/M systems do not have interrupt driven I/O. Function keys may not work as expected.

- The variable **CCtlCk** controls the number of times the console is re-tried for console input after one character is read. Its default value is 6. Change it to any number from 1 to 255, as follows:

```
extern char CCtlCk;
CCtlCk = 12;
```

  If you have trouble with function keys, then this variable may have to be changed to a higher number. Especially true for slower terminals and CPU speeds under 4mhz.

- In Cmode <> 1, any character is acceptable, including NULL. The method used for character mode is direct BIOS with manual echo in Cmode = 0. We assume that the BIOS can be found by reading the address at 0x0001.

- In Cmode = 1, the usual CP/M editing functions are active, plus ctrl-P, ctrl-S and ctrl-C. In addition, ctrl-B will cause a jump to the ctrl-B processor CtlB (). A ctrl-C will cause an exit to CP/M, regardless of its entry position on the line.

The chown Function

Purpose:

Change the owner of an existing file.

Function Header:

```
int chown(name,owner,group)

char *name;             Ascii file name, CP/M conventions

int  owner,             Owner of the file, an integer known
                        to Unix via the login file.
     group;             Unix group number, an integer that
                        Unix looks up at login.
```

Both the owner number and the group number are ignored by CP/M.

Returns:

0	File exists
-1	File not found

Notes:

- o CP/M treats this call as a file look-up only. The user of a CP/M system is the owner.
- o Chown() makes Unix source code compile without errors, even though it may not run properly.
- o Unix calls getuid(), getgid() can look up the owner and group numbers, while setuid() and setgid() are capable of setting the numbers.

The clearerr Function

Purpose:

Clears a stream error. This function does nothing under C/80.

Function Header:

```
int clearerr(fp)
FILE *fp;           Open stream pointer.
```

Returns:

Nothing.

Example: Compile Unix code under C/80.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int x;
     while((x = getchar()) != EOF) {
          if(ferror(stdout)) break;
          putchar(x);
     }
     clearerr(stdout);
     fputs("Operation complete\n",stdout);
}
```

Notes:

o clearerr() is a no-op under C/80 because no provision has been made to survive a disk output error.

The close Function

Purpose:

Flushes the file buffer to disk, writes the file control block to the disk directory and releases the file descriptor.

Function Header:

```
int  close(fd)
int  fd;                    File descriptor
```

Returns:

```
 0       It worked
-1       It failed
```

Example:

Close a file that was opened by open() and written to by write().

```
#include <unix.h>
#include <stdio.h>
main()
{
int fd;
  fd = open("MYFILE",1);
  if(fd) {
    write(fd,"This is a test",14);
    close(fd);
  }
}
```

Notes:

- Requires a file descriptor as argument.
- Uses fclose() .
- The fflush() done by fclose() is a no-op in case no stream I/O was done.

The cmpi, cmpl, cmpf and cmps Functions

Purpose:

Compare integer, long integer, float and string data for qsort(). These are not library functions but rather source code segments ready to insert into your application. Often seen as code macros (C/80 can't use code macros).

Function source code:

```
int cmpi(n,m)
int *n,*m;                  Pointers to 16-bit integer data.
{ return (*n - *m); }

int cmpl(n,m)
long *n,*m;                 Pointers to 32-bit integer data.
{ return (*n - *m); }

int cmpf(n,m)
float *n,*m;                Pointers to 32-bit float data.
{ return (*n - *m); }

int cmps(n,m)
char *n,*m;                 Pointers to strings, null-terminated.
{ return (strcmp(n,m)); }
```

Returns:

For numerical compares:

```
  0             *n == *m            (same as strcmp)
< 0             *n < *m
> 0             *n > *m
```

For string compares, the return is the same as for strcmp(), which in turn is the same as the above, except for interpretation.

Example: Use qsort() to sort integer data.

```
#include <unix.h>
#include <stdio.h>
int q[10] = {12,1,3,-2,16,7,9,34,-90,10};
int p[10] = {12,1,3,-2,16,7,9,34,-90,10};
main()
{
int i;
  qsort(p,10,2,cmpi);
  for(i=0;i<10;++i) printf("%d. %d, %d\n",i,p[i],q[i]);
}
```

Program output

```
0. -90, 12
1. -2, 1
2. 1, 3
3. 3, -2
4. 7, 16
5. 9, 7
6. 10, 9
7. 12, 34
8. 16, -90
9. 34, 10
```

Notes:

- The pointer usage is required in qsort().

- To overcome unterminated string problems, use a special cmps():

```
int length = 10;
cmps(n,m)
char *n,*m;
{
int i;
   for(i=0;i<length;++i)
     if(n[i]-m[i]) return (n[i]-m[i]);
   return 0;
}
```

The problem that this special compare overcomes is presented by strings that are not null-terminated, but have a fixed length.

The comp Function

Purpose:

Compares two strings for a substring match beginning at the first character of the first string.

Function Header:

```
int comp(str1,str2)
char *str1,*str2;              Source strings, null-terminated.
```

Returns:

```
 0  if str1 extends str2 (str2 is a substring)
-1  if str2 is longer than str1
 J  if str2 mismatches str1 at position J
```

Example: Find the word SAILOR inside a sentence typed by the user.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i;
     if(argc <= 1) exit();
     for(i=0;argv[1][i] != '\0';++i) {
       if(comp(&argv[1][i],"SAILOR")==0) {
          printf("Word SAILOR found at index %d\n",i);
          return;
       }
     }
     printf("Word SAILOR not found in string\n");
}
```

Notes:

- comp() is similar to the BASIC function instr(A$,B$). It takes two arguments.
- The value of comp("abcd","abc") is 0, because str2 = "abc" is a substring of str1 = "abcd"
- The value of comp("xabcd","abc") is 1 because the mismatch is at the first string position (1, not 0).
- The value of comp("abcd","abcabc") is -1 because the second string is longer than the first string.

The conout and con Functions

Purpose:

Conout() prints multiple-argument characters to the console device. No re-direction. Used primarily as a debugging tool or as fast output in graphics routines.

Con() is the driver routine for conout. It prints one character and takes one argument.

Function Header:

```
int conout(c1,c2,...)
char c1,c2,...              Characters to be printed.

int con(c)
char c;                     Character to be printed.
```

Returns:

Nothing useful. The printing is done by a standard BDOS function.

Example: Print escape codes to the Z29 screen to enable inverse video graphics characters to be displayed.

```
#include <stdio.h>
main()
{
     conout(27,'p',27,'F');
     conout('g','r','a','p','h','i','c','s',' ','t','e','s','t');
     conout(27,'G',27,'q');
}
```

Notes:

- Very low overhead. Recommended for ROM applications where the essential functions of putchar() are required.
- The symbol conout() is defined in STDIO.H. The word conout is not a library global symbol. Without the #define information in STDIO.H, this function does not exist.
- The word CON is a global symbol in the library. It does not require #define information to be accessible.
- Not to be confused with putc() or putchar(), which allow re-direction. Conout() and con() work only with the console device.

The Copen_ Function

Purpose:

Opens streams stdin and stdout prior to the startup code which calls the function main().

Function Header:

```
int Copen_()
```

Returns:

0 Always.

Example: Making a new Copen_() to turn off re-direction.

```
Copen_()
{
    return 0;
}
```

Notes:

- The provided Copen_() routine opens files for streams stdin and stdout using descriptors fin and fout, which are of cast extern int. The file names are obtained from char *p_fin,*p_fout, which were loaded with the proper pointers by the routine tokens().
- The standard way to handle re-direction failures is to fall into the abort exit. This is the method presently used by Copen_() in the main library.
- See also Cexit_(), which closes stdin and stdout streams during the standard C_exit() routine.

The core Function

Purpose:

Obtains contiguous memory from the system and sets the upper bound of the program+data area. Aborts with an error message on failure.

Function Header:

```
char *core(n)
int n;              Amount of memory requested. The maximum value
                    of n is 32767. See notes below.
```

Returns:

addr	on success, addr = base address of the contiguous area of memory n bytes in length.
system	Failure. Prints "Out of Memory" and warm boots.

Example: Get a buffer, fill it with nulls.

```
#include <stdio.h>
main()
{
char *p,*core();
     p = core(4096);          /* warm boot on failure */
     fillchr(p,'\0',4096);
}
```

Example: Read a file by re-direction into heap space, copying the heap address to an array of character pointers. Print to console when done.

```
#include <stdio.h>
main()
{
int i,n;
char *p[1024];
char s[129];
char *core();
     n = 0;
     while(getln(s,128) != EOF) {
          if(n >= 1024) break;
          strcpy(p[n++] = core(strlen(s)+1),s);
     }
     for(i=0;i<n;++i) puts(p[i]);
}
```

Notes:

- This function will work with n an unsigned integer. There is no failure recovery, however. The penalty is warm boot.

The cos Function

Purpose:

Compute the cosine of real float value x. The value x is assumed in radians (not degrees).

Function Header:

```
float cos(x)            Cosine.
float x;                Float value for computation.
```

Returns:

```
number          Answer between -1 and 1
                No error checking for large x
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float cos();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("cos(%f) = %f, errno = %d\n",x,cos(x),errno);
}
```

Notes:

o The method used is cos(x) = sin(PI/2 + x).

o Use deg() and rad() for conversions.

The cosh Function

Purpose:

Compute the hyperbolic cosine of real float value x.

Function Header:

```
float cosh(x)       Hyperbolic cosine (e^x + e^-x)/2.
float x;            Positive, negative or zero argument.
```

Returns:

```
number          x in range
INF             x too large positive (see notes).
-INF            x too large negative (see notes).
                errno = ERANGE returned to flag error.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float cosh();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("cosh(%f) = %f, errno = %d\n",x,cosh(x),errno);
}
```

Notes:

o Method used is cosh(x) = (exp(x) + exp(-x))/2.0.

o The value of EXPLARGE was found by solving the equation exp(x) = 1.0e39.

The creat, creata, creatb Functions

Purpose:

Add a new file to the disk directory. The file can be referenced by descriptor fd but not as a stream file.

Function Header:

```
int  creat(name,pmode)          Ascii create file
char *name;                     Null-terminated file name
int  pmode;                     Unix protection mode, ignored

int  creata(name,pmode)         Same as creat(), see unix.h
char *name;
int  pmode;

int  creatb(name,pmode)         Binary create file
char *name;
int  pmode;
```

Returns:

```
fd      File descriptor for the opened file
-1      It failed
```

Example:

Create a binary file and write a buffer of raw data.

```
#include <unix.h>
#include <stdio.h>
main()
{
char buf[128];
int fd;
  fd = creatb("JUNK",0);
  if(fd) {
    fillchr(buf,'\7',128);
    writeb(fd,buf,128);
    close(fd);
  }
}
```

Notes:

o creata() is update mode, ascii.

o creatb() is update mode binary.

o creat() is defined only in the unix.h header file.

The CtlB_ Interrupt Function

Purpose:

Run a user subroutine each time ctrl-B is intercepted. Often this routine exits to warm boot or sets an exit flag from a deeply buried sequence of function calls. May be used in conjunction with setjmp() and longjmp().

Function Header:

```
int CtlB ()          Invoke user subroutine manually.
char *CtlB;          Storage for user subroutine address.
                     This address is initially set to warm boot.
```

Returns:

Return value of the user subroutine.

Example: Set up a break condition in a subroutine. This routine sets up a function myctlb() to allow the user to break out of the I/O loop by typing ctrl-B.

```
static int flag = FALSE;

myctrlb()
{
     flag = TRUE;
}

int fileprint()
{
char *p;
extern char *CtlB;
     p = CtlB;
     CtlB = myctrlb;
     puts("Enter ctrl-B to quit");
     puts("Printing...");
     while(flag == 0) {
          if(g()) break;       /* print a line */
          CtlCk();             /* check ctrl-B   */
     }
     CtlB = p;
}
```

Notes:

- The routine CtlCk() intercepts type-aheads and puts them into the type-ahead buffer at Cbuf. If Cmode == 1, then a break character test is performed. Ctrl-B runs CtlB_().
- You can manually run CtlB_(). No one ever does this, however. The ctrl-B processor is a kind of limited interrupt, which is enabled by the programmer and checked only when it makes sense to check for a user interrupt.
- CtlCk() is called before each disk access for a read or write. Therefore, CtlB_() may be called without the programmer's knowledge.

The CtlCk Function

Purpose:

Check for ctrl-C and ctrl-B at run time.

Function Header:

```
int CtlCk()
```

Returns:

In any case, available characters are added to the console buffer. In line mode (Cmode = 1) ctrl-C causes an abort exit through a exit(), while ctrl-B causes a jump to function Ctl_B (). In character mode (Cmode = 0 or 2) the abort keys are ignored and processing continues. The character is echoed in Cmode = 0.

Example: Check for ctrl-C while copying from stdin to stdout. Most common re-direction is disk for stdin and line printer for stdout.

```
#include <stdio.h>
main()
{
int x;
  while(TRUE) {
       x = getchar();
       if(x == EOF) break;
       CtlCk();
       putchar(x);
  }
}
```

Notes:

- The internal operation of CtlCk() is to call the keyboard status routine to see if the user typed a character. If so, then the character is read no-echo and added to the console buffer.
- It is common to call CtlCk() during character output to devices. During disk reads and writes, CtlCk() is called before each disk access, automatically (see WRITE.CC).
- In character mode, a call to CtlCk() is the same as adding available characters to the console buffer. No special action is taken for ctrl-C or ctrl-B.

The cvupper Function

Purpose:

Converts a string to upper case only, skipping over those characters that do not qualify for conversion.

Function Header:

```
char *cvupper(s)
char *s;                    String to convert, null-terminated.
```

Returns:

The string address s, with the characters of the string converted to upper case, as appropriate. Only lower case a-z are converted.

Example: Convert an input line to all upper case.

```
#include <stdio.h>
main()
{
char s[129];

     printf("Enter a line of text: ");
     gets(s);
     cvupper(s);
     printf("UPPERcase line:\n%s\n",s);
}
```

Notes:

- This function is unlikely to exist in other libraries, but it is easily written from toupper(). It is suggested that you bury the uppercase conversion in cvupper().
- Cvupper() is an honest function, it is not a macro, and it has no side effects.

The C_cmln Function

Purpose:

Copies the default buffer at 0x80 to the C console buffer located at (char *)Cbuf. Also copies the running program name from the CCP buffer area, if possible.

Function Header:

```
VOID C_cmln()
```

Returns:

Nothing. Does a buffer copy.

Example: Making a new C_cmln() to disable command line copying.

```
C_cmln()
{
     return 0;
}
```

Notes:

- To copy the command line but leave out the copying of the running program name, use n=peekb(0x80) to find out the length of the default buffer and then use strncpy(Cbuf,0x81,1+n).
- One function of the current C_cmln() is to copy the name of currently running program from the CCP buffer. This may or may not be successful, especially if program chaining is being used. Also beware if the CCP size is not standard.

The decodF Function

Purpose:

Decodes a file control block into a printable ASCII string.

Function Header:

```
char *decodF(fcb)
char fcb[13];            File control block section.
```

Returns:

```
(char *)p          p = address of a static area in memory where
                   the decoded string is located.
```

Example: Print out the default file control blocks at 0x5C, 0x6C.

```
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
char *decodF();
  insert("FILE1.TXT FILE2.TXT");
  printf("File name from FCB at 0x5C = %s\n",decodF(0x5C));
  printf("File name from FCB at 0x6C = %s\n",decodF(0x6C));
}
```

Notes:

- The buffer for decodF() is re-used on each call. If you need to keep the decoded name around for a while, then copy it to safety.
- Only the first 13 characters of the file control block are accessed during the call.
- There is no bdos() or bios() call that simulates this function.

The deg Function

Purpose:

Changes float radians x to degrees.

Function Header:

```
float deg(x)            Conversion to degrees.
float x;                Value in radians to convert.
```

Returns:

angle in degrees (a FLOAT)

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
float atof();
float deg();
     printf("Enter float x in radians: "); gets(s); x = atof(s);
     printf("deg(%f) = %f\n",x,deg(x));
}
```

Notes:

- Method used is angle = (180.0/PI)*x.
- No error codes returned. No need to code for errno.
- No overflow check.
- No auto conversion to FLOAT.

The dsort Function

Purpose:

Distribution sort in High-Speed Assembler. Implements Donald Knuth's MathSort algorithm for a single digit. Multiple calls are required to complete the sorting. Best for byte, integer, long and float data types. Stable.

Function Header:

```
struct mathsort {
int
size,        integer number of records to sort
offset;      integer offset in bytes to target byte (+ or -)
char
*input,      address of input table
*output,     address of output table
*count;      address of counter table
};

VOID dsort(table)
struct mathsort *table;
```

Returns:

Nothing. The output table is changed to reflect the new sorted order. The actual data in memory is unchanged.

Example: Sort a list of long integers.

```
#define pfLONG 1
#include <stdio.h>
main()
{
int i;
char *tmp;
static
struct mathsort table;
auto
char *data_in[10],*data_out[10];
auto
int scratch[256];
static
long ldata[] = {
     9349324,213232,343243,655666,546554,
     3454445,454544,565421,324221,344343
};
long peekl();

     for(i=0;i<10;++i) data_in[i] = &ldata[i];
     table.input = data_in;
     table.output = data_out;
     table.count = scratch;
     table.size = 10;
     table.offset = 0;

     for(i=0;i<4;++i) {
     dsort(&table);                    /* sort one digit   */
     tmp = table.input;                /* swap the input   */
     table.input = table.output;       /* and output       */
     table.output = tmp;               /* tables, then     */
     ++table.offset;                   /* increment offset */
     }

     for(i=0;i<10;++i) printf("%ld\n",peekl(data_in[i]));
}
```

Notes:

- The order of the arguments is essential.
- The **input table** is an array of addresses of strings, which need not be null-delimited. Generally, the strings are numbers. The algorithm is stable, permitting sorts of byte, integer, long, float and double data types. The **output table** is the sorted order of the input table.
- Only one digit is processed on each call. To process a number of digits, write a loop which swaps the input and output tables. See the example above for method.

The dsort16 Function

Purpose:

Distribution sort for 16-bit integers in High-Speed Assembler. Implements Donald Knuth's Algorithm D (MathSort).

Function Header:

```
struct mathsort {
int
size,        integer number of records to sort
offset;      integer flag, presently ignored. See notes.
char
*input,      address of input table
*output,     address of output table
*count;      address of counter table
};

VOID dsort16(table)
struct mathsort *table;
```

Returns:

Nothing. The input table is changed to reflect the new sorted order. The actual data in memory is unchanged. The output table and the counter table return no information.

Example: Sort a list of integers.

```
#include <stdio.h>
main()
{
int i;
static struct mathsort table;
auto char *data_in[10],*data_out[10];
auto int scratch[256];
static int data[] = {
     9324,3232,3243,5666,6554,
     4445,4544,5421,4221,4343
};
unsigned peekw();
     for(i=0;i<10;++i) data_in[i] = &data[i];
     table.input = data_in;
     table.output = data_out;
     table.count = scratch;
     table.size = 10;
     dsort16(&table);
     for(i=0;i<10;++i) printf("%u\n",peekw(data_in[i]));
}
```

Notes:

- The order of the structure arguments is essential. The routine accepts the ADDRESS of a structure table.
- The MathSort algorithm is stable. In particular, duplicates are not moved relative to one another.
- The **input table** is an array of addresses of integer data. The **output table** is the sorted order of the input table. Since two passes and a swap of in and out tables occurs, the sorted order appears in the input table when the routine dsort16() returns.
- The actual integer data is elsewhere in memory, since the **input** array contains just the addresses of the data, not the data itself. On return, the **input** array contains the addresses of the integer data, in sorted order. The counter table and the output table are used as scratch space, hence contain nothing useful.
- To sort 32-bit data with Dsort16(), use it once to sort the first 16 bits, then increment all the input table pointers by 2 and call Dsort16() again. Decrement all the input table pointers by 2, and the input table gives the sorted order.
- Source code is in 8080 assembler. Algorithm in C is given in the source code comments. See the source archives.

The edata, end and etext Variables

Purpose:

Defines end, etext, edata locations in a C program. These are external character pointer variables.

Function Header:

```
extern char *end, *etext, *edata;
```

Example: Print the data locations in a C program.

```
#include <unix.h>
#include <stdio.h>
main()
{
extern char *end, *etext, *edata;
  printf("end=%u, etext=%u, edata=%u\n",end,etext,edata);
}
```

Notes:

- o **end** The next usable address after program load. Marks the end of the file buffer area. This address is returned by sbrk(0).
- o **etext** The physical end of the program text before the file buffers and FCB buffers begin. The address contained here is used in IOTABLE.CC to define the file buffers. See the archives.
- o **edata** Same as etext for C/80 because code and data are in the same place.

The errno Variable

Purpose:

Error return variable for the transcendental functions. Also used for system error returns.

Function Header:

```
extern int errno;        This variable is a global in CLIB.REL
```

Returns:

The system error number.

Example: Test for an error return after using the sine function.

```
#include <math.h>
#include <stdio.h>
main()
{
static float f = 1.0123;
float sin();
extern int errno;
     errno = 0;
     printf("sin(%f) = %f, errno = %d\n",f,sin(f),errno);
}
```

Notes:

- Use perror() to print an error string for errno.
- The variable in CLIB.REL will be used if you don't use the symbol errno in your program.
- A compile error will occur if you don't declare errno as extern or as a global variable (outside any function and not static).

The error Function

Purpose:

Prints multiple-argument strings to the console device. No redirection. Used primarily as a debugging tool.

Function Header:

```
VOID error(str1,str2,...)
char *str1,*str2...          Source strings to be printed.
```

Returns:

Nothing useful. The printing is done by a direct console function that acts through BDOS.

Example: Print error messages in a program that will be burned into EPROM at a later date. Uses setjmp() and longjmp() to break out of deeply buried control loops.

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf  env[1];
main()
{
   switch(setjmp(env)) {
   case 0: break;
   case 1: error("Error 1 from InitSystem\n"); break;
   case 2: error("Error 2 from WakePorts"); break;
   case 3: error("Error 3 from MainLoop\n","Dead solenoid\n");
           break;
   case 4: error("Error 4 from MainLoop\n","Bad timing\n"); break;
   case 5: error("Error 5 from MainLoop\n","Bad robotics\n");
           break;
   }
   InitSystem();
   WakePorts();
   MainLoop();
}
```

Notes:

- Very low overhead. Recommended for ROM applications where real estate is at a premium but some of the functions of printf() are required.
- The symbol error() is defined in STDIO.H. The word error is not a library global symbol.
- Not to be confused with ferror or perror.

The exit Function

Purpose:

Standard exit. Flushes buffers and closes the file control block to the disk directory for stream stdout.

Function Header:

```
VOID exit()
```

Returns:

Nothing. Exits to warm boot.

Example: Use of exit().

```
#include <stdio.h>
main()
{
     printf("Standard C exit\n");
     exit();
}
```

Notes:

- Calls the routine Cexit_(), then falls into the abort exit.
- If the Cexit_() routine is the one provided, then buffer flushing and closing will occur for at least stream stdout. But any other open streams will be ignored. Open files will leave orphan disk directory entries with 0 records.
- To change the character of the standard exit requires a new routine Cexit_(). Most systems flush all buffers and close all file control blocks back to the disk directory. See Cexit_() for an example of how to re-write it to simulate such a system.

The exp Function

Purpose:

Compute the exponential of real float value x

Function Header:

```
float exp(x)        Power with base e = 2.718 and exponent x.
float x;            Positive, negative or zero argument.
```

Returns:

```
number          x in range
INF             x too large positive (see notes).
0               x too large negative (see notes).
                errno = ERANGE returned to flag error.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float exp();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("exp(%f) = %f, errno = %d\n",x,exp(x),errno);
}
```

Notes:

- Method used is exp(x) = pow(EBASE,x) where EBASE = 2.71828 = e = base for natural logarithm, approximately.
- The error range is found by solving exp(x) = 1.0e39.

The expand Function

Purpose:

Expands wild card file names on the command line into a complete list. The list is passed to main() by the usual argc, argv[] method.

Function Header:

```
expand(a,b)
int *a;              Address of argc variable
                     Usually you insert &argc
char *b;             Address of array argv[]
                     Usually you use &argv
```

Returns:

New argc value, old value is lost.
New table *argv[] with expanded wild cards.
Duplicates from expansion are ignored.

Example: Normal wildcard expansion in a C program. This example displays the wildcard expansions.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
 int i;
   expand(&argc,&argv);
   for(i=0;i<argc;++i)
     printf("argv[%d] = %s\n",i,argv[i]);
}
```

Example: Wildcard expansion of a command line inside a program. Commas or blanks may separate names. Note &-operator usage and the extra pointer char **bn.

```
#include <unix.h>
#include <stdio.h>
main()
{
 int i;
 int a;
 char buf[129];
 char *b[128];
 char *p,*name,**bn;
 char *sob(),*fnb();

    b[0] = "Unused"; a = 1; p = name = buf;
    printf("Enter wildcard names: "); gets(buf);
    while(*p) {if(*p == ',') *p = ' '; ++p;}
    do {
        name = sob(name);
        if(name[0]) b[a++] = name;
        p = fnb(name); if(p[0] == '\0') break;
        p[0] = '\0'; name = p+1;
    } while(*name);
    bn = b;
    expand(&a,&bn);
    for(i=0;i<a;++i)
        printf("b[%d] = %s\n",i,bn[i]);
}
```

Example: Break a command line into tokens using the tokens() routine in the library. No wildcard expansion.

```
#include <stdio.h>
main()
{
 int i,a;
 char buf[129];
 char *b[100];

    printf("Enter a C-style command line\n: ");
    gets(buf); if(buf[0] == '\0') exit();
    b[0] = b[1] = "Not in use";
    a = tokens(b,buf);
    printf("Re-direction files: %s, %s\n",b[0],b[1]);
    for(i=0;i<a;++i)
        printf("b[%d] = %s\n",i+2,b[i+2]);
}
```

Notes:

- It can run out of room. Uses sbrk() to get the space for the expansions.
- Takes 600 bytes of stack space.
- CP/M-80 lets you access the command line from the default buffer. Copy it to safety with: strncpy(buffer,129,peekb(128));
- The above won't work with re-direction because C/80 uses the default buffer to open the < & > files before running main.
- See also TOKENS.C and the tokens() routine for a method of decoding a C command line.
- Complete source code for expand(). It is often the case that you need a similar expansion function, with minor changes. Here is food for thought:

```
static int nomatch(n,f,t)
int n; char *f[],*t;
{
    while(--n) {if(strcmp(f[n],t)==0) return 0;} return 1;
}
#define MAXFILES        255     /* max number of expansions */
expand(argcp,argvp)
int *argcp; char **argvp;
{
  char *fn[MAXFILES];
  static int j,nf,c;
  static char **v,*arg,*p;
  char *core(),*decodF(),*strcpy(),*index();
  char *findfirst(),*findnext();

    c = *argcp; v = *argvp; fn[0] = v[0];
    for (nf=1,j=1;((nf < MAXFILES) && (j < c)); ++j) {
        cvupper(arg=v[j]);
        if(index(arg,'?') || index(arg,'*')) {
            p = findfirst('\0',arg);
            while (p != (char *)(-1)) {
                p = decodF(p);
                if(nomatch(nf,fn,p))
                    strcpy(fn[nf++]=core(1+strlen(p)),p);
                p = findnext();
            }
        }
        else if(nomatch(nf,fn,arg)) fn[nf++] = arg;
    }
    *argcp = nf; fn[nf++] = -1;
    v = *argvp = core(sizeof(char *) * nf);
    while(nf--) v[nf] = fn[nf];
}
```

The fabs Function

Purpose:

Compute the absolute value of a float number.

Function Header:

```
float fabs(value)           Float Absolute value
float value;                Float argument, 32 bits
```

Returns:

The absolute float value of the argument.

Example: Print the float absolute value of a floating point number entered at the console.

```
#define pfFLOAT 1
#include <stdio.h>
main()
{
char s[129];
float f,fabs(),atof();

     printf("Enter float number f: ");
     gets(s); f = atof(s);
     printf("fabs(%f) = %f\n",f,fabs(f));
}
```

Notes:

- Not a macro. But other libraries will use one. Watch out when you write your code, and when you port it to other machines.
- There are no side effects. This is an honest function.
- The cast must appear explicitly in your program. For example, if you use fabs(), then include a declaration of the form

  ```
  float fabs();
  ```

- To print out floats using printf(), employ the header file switch called pfFLOAT. See the example above.
- A common error is to write g = abs(f) where f,g are floats. This has undefined results with no compile-time error reporting.

The fclose Function

Purpose:

Flush the stream buffer to disk. Close the file control block to the disk directory. Remove the stream pointer from system tables.

Function Header:

```
int fclose(fp)
FILE *fp;          Open stream pointer.
```

Returns:

```
EOF          On error.
0            Success.
```

Example: Close the stream stdout.

```
/* Command line is A>main >foo.bar */
#include <stdio.h>
main()
{
     printf("This line is sent to stream stdout\n");
     fclose(stdout);
}
```

Notes:

- An fclose() operation will fail if the buffer flush does not complete due to a disk error or lack of disk space. It can also fail due to a write-protect tab.
- An fclose() on a device sends an end of file character in case the device requires one. This is true for the printer LST:. It gets ctrl-L.
- An fclose() on the console is a NOP.
- Streams with descriptors 252,253,254,255 are devices. They are treated differently than files during I/O because there is no buffering.
- Programs written with re-direction in force always call Cexit_() to close stdout. The close is done by fclose().

The fdopen Function

Purpose:

Open a stream file using an existing file descriptor. Applies to disk files only. No devices allowed.

Function Header:

```
FILE *fdopen(fd,mode)
int fd;                 Open file descriptor
char *mode;             Mode from fopen().
```

Returns:

```
0           Open failed
fp          Open worked, fp=stream pointer
```

Example: Create a TMP file by descriptor, then re-open as a stream.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int fd;
FILE *fp,*fopen();
     fd = creat("TMP",0);      /* protection mode ignored */
     if(fd != -1) {
       fp = fdopen(fd,"w");
       fputs("It worked\n",fp);
       fclose(fp);
       puts("TMP created");
     }
     else puts("Open failure on TMP");
}
```

Notes:

- In this library, all files that use file descriptor tables are also buffered files. Do not expect this feature on other systems.
- The function fdopen() can change the access mode of the file.
- Fdopen() uses a call to freopa(). Since the latter can handle ascii and binary mode changes, no further code is needed.

The feof Function

Purpose:

Tests for end of file on a stream.

Function Header:

```
int feof(fp)
FILE *fp;          Open stream pointer.
```

Returns:

```
0              Not at end of file
nonzero        Stream at end of file
```

Example: Check for end of file when using getw().

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
     while(TRUE) {
       if((x = getw(stdin)) == EOF) if(feof(stdin)) break;
       putw(x,stdout);
     }
}
```

Notes:

- Testing putc() is impossible, since C/80 normally aborts when it runs out of disk space.
- Files opened for write are at end of file, unless seek() has been used, in which case the use of feof() makes no sense at all.
- The best way to understand feof() is to study its source code:

```
int feof(fp)
FILE *fp;
{
static int x,fd;
extern char IOmode[];
   fd = fileno(fp);
   if(IOmode[fd]=='w' || (x = getcbinary(fp)) == EOF ||
      (x == 26)) return 1;
   ungetc(x,fp);
   return 0;
}
```

The ferror Function

Purpose:

Tests for a stream error. This function returns 0 under C/80.

Function Header:

```
int ferror(fp)
FILE *fp;           Open stream pointer.
```

Returns:

```
0              Always.
```

Example: Compile Unix code under C/80.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
int x;
     while((x = getchar()) != EOF) {
          if(ferror(stdout)) break;
          putchar(x);
     }
}
```

Notes:

o ferror() is a no-op under C/80 because no provision has been made to survive a disk output error.

The fflush Function

Purpose:

Writes all buffered data onto the disk without changing file pointers or disturbing character I/O functions. Writes the current file control block into the disk directory, but leaves the file open for I/O.

Function Header:

```
#include <unix.h>          Service is #define fflush uflush
int fflush(fp)
FILE *fp;                  Stream pointer for a disk file
```

WARNING: The header file UNIX.H defines fflush as uflush. The main library has a function called fflush(), which does the same flush operation, but fails to update the disk directory or check for devices. If you want to use both, then **#undef fflush** and call each by its internal library name, fflush or uflush.

Returns:

0	Success
-1	Failure

Example: Flush random file block to disk during I/O wait.

```
while(TRUE) {
  if(bdos(11,0)) return getchar();
  if(dirtyblock) { fflush(fp); dirtyblock=0;}
}
```

Notes:

- The uflush() function writes the disk buffer to disk and then the file control block to disk.
- The main library fflush() does not check for devices. It does not write the file control block back into the disk directory. The symbol FFLUSH is retained for old software compatibility.
- New software should use UNIX.H. Direct reference to uflush() should be avoided.
- Under C/PM 2.2, devices like the console and printer are not files, so uflush() does not act on these devices. Devices do not need flushing under CP/M because they are not buffered. When you print to the console or printer, it happens a byte at a time.

The fgets Function

Purpose:

Reads up to n characters from an open stream and null-terminates the string. Breaks before n characters if end of file or a newline is encountered.

Function Header:

```
char *fgets(s,n,fp)
char *s;                Base address of the storage area.
int n;                  Maximum size of storage area in bytes.
FILE *fp;               Open stream pointer.
```

Returns:

```
(char *)0               End of file and string empty.
(char *)0               Stream error and partially filled string.
(char *)s               Success, base of storage area.
```

Example: Echo lines typed at the console until ctrl-Z.

```
#include <stdio.h>
main()
{
char s[129];
     while(TRUE) {
          fprintf(stderr,"Enter a string or ctrl-Z to exit: ");
          if(fgets(s,40,stdin) == (char *)0) break;
          fprintf(stderr,"%s",s);
     }
}
```

Notes:

- It is impossible to tell whether a null pointer return stands for an error or end of file.
- The array s[] may be filled with invalid data in case of an error.
- Newlines read from the stream are appended to the array s[]. This is different from gets(), which strips the newline on input. In practise, it means puts() is not useful for input obtained from fgets().

The fileno Function

Purpose:

Returns the file descriptor for a stream.

Function Header:

```
int fileno(fp)
FILE *fp;           Open stream pointer.
```

Returns:

fd File descriptor for stream fp.

In the C/80 implementation, fd is the device handle 252, 253, 254 or 255, or fd is in the range 1..MAXCHN-1, which represents the array index which accesses the file information in IOTABLE.

Example: Use read() on an open stream fp.

```
#include <unix.h>
#include <stdio.h>
main()
{
char *buf,buffer[128];
FILE *fp,*fopen();
int fd,n;

  fp = fopen("MYFILE","r");
  if(fp) {
    buf = buffer;
    fd = fileno(fp);
    n = read(fd,buf,128);
    while(n--) putchar(*buf++);
    fclose(fp);
  }
}
```

Notes:

- Use fileno() to maintain Unix System III compatibility.
- While C/80 treats streams and descriptors as the same object, other systems do not.
- Read() and write() use descriptors rather than streams. This area of distinction accounts for most compile-time errors for older C/80 source code using Unix-style compilers.

The fillchr Function

Purpose:

Fills a region of memory with a byte value.

Function Header:

```
char *fillchr(dest,c,n)
char *dest;                     Destination address.
char c;                         Byte value for fill.
int n;                          Number of bytes to fill.
```

Returns:

```
dest+n                          Next location after fill.
```

Example: Fill a buffer with nulls.

```
#include <stdio.h>
main()
{
char *p,*fillchr();
char s[100];
     p = fillchr(s,'\0',100);
     printf("s = %u, fillchr(s,'\0',100) = %u\n",s,p);
}
```

Notes:

- Some libraries call this function setmem(). The ordering of the arguments is different, however. We use the same ordering as is used by strncpy() and strncat(). It looks like setmem() has assembly language origins for an Intel CPU. It is not a Bell Labs Unix V7 function, so let the argument continue.

- The integer n could be unsigned, because this function does its arithmetic using unsigned compares. For portability we recommend restraining its argument to type integer.

- PASCAL has a function called fillchar() that is similar to the one implemented in this library. It accounted for glowing claims about the speed of PASCAL over C when the first BYTE Sieve of Eratosthenes Benchmark was published (Sept, 1981).

The fin and fout Variables

Purpose:

File descriptors fin and fout for streams stdin and stdout.

Example: Display the file descriptors for the standard input and output streams. Unix-compatible calling sequence.

```
#include <unix.h>
#include <stdio.h>
main()
{
  printf("fin = %d, fout = %d\n",fileno(stdin),fileno(stdout));
}
```

Example: Display the file descriptors for the standard input and output streams. Direct method.

```
#include <stdio.h>
main()
{
extern int fin,fout;
  printf("fin = %d, fout = %d\n",fin,fout);
}
```

Notes:

- This library simulates stream pointers in STDIO.H by means of special functions fin__() and fout__().
- The function fileno() returns a file descriptor for a given open stream pointer.
- Descriptors fin and fout are both 0, which refers to the console device, unless re-direction is in force.
- The naming conventions for the descriptors comes from obsolete versions of Unix. Beware in using fin and fout when trying to write portable code.

The fin__ and fout__ Functions

Purpose:

Return the file descriptors for streams stdin and stdout.

Function Header:

```
int fin__()
int fout__()
```

Returns:

fin__()	The file descriptor for stream stdin
fout__()	The file descriptor for stream stdout

Example: Define macros for streams stdin and stdout.

```
#define FILE int
#define stdin (FILE *)fin__()
#define stdout (FILE *)fout__()
```

Notes:

- The above macros are implemented in STDIO.H.
- Meaningless statements like stdin = 0 will hopefully produce compiler error messages.
- The fact that fin and fout are extern variables will cause compile errors, unless they are declared properly. You can usually use fin__() and fout__(), thereby avoiding the extern declarations.
- The functions fin__() and fout__() present a portable interface across the versions of Unix. It is best not to use these functions at all. However, if you must, then it is better to bury the problems in fin__() and fout__() than to directly use the externs fin and fout.

The findFIRST Function

Purpose:

Finds a disk file by name or wildcard file spec.

Function Header:

```
char *findFIRST(extent,filename)
char extent;                    Extent, '\0' is normal.
char *filename;                 Name of file, null-terminated.
```

Returns:

```
(char *)-1          Failure
(char *)p           p = address of matching file control block
                    taken from the disk directory. Actually, p
                    points into the default buffer at 0x80.
```

Example: Search for a wildcard file name and print the first one found.

```
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
char *p;
char *decodF(),*findFIRST();
     if(argc <= 1) exit();
     if((p = findFIRST('\0',argv[1])) != -1)
          printf("First match = %s\n",decodF(p));
     else printf("File not found\n");
}
```

Notes:

- Decodf(fcb) decodes a file control block into a printable ASCII file name in the usual CP/M format.
- An extent of '?' will search all user areas.
- A file name of "*.*" will search for all files. The question mark wildcard also works.
- This function uses the default buffer at BOOT+0x80. In most applications you have to copy the file control block to safety or else use decodF() and copy the return string to safety.
- The findFIRST() function can be simulated on other systems by using makeFCB(fcb,file) and bdos(17,fcb). The latter returns the directory entry offset (mod 32) into the default buffer.

The findNEXT Function

Purpose:

Finds the next file name match after a call to findFIRST().

Function Header:

```
char *findNEXT()
```

Returns:

(char *)-1	Failure
(char *)p	p = address of matching file control block taken from the disk directory. Actually, p points into the default buffer.

Example: Search for a wildcard file name and print all occurrences.

```
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
int i;
char *p;
char *decodF(),*findFIRST(),*findNEXT();
     if(argc <= 1) exit();
     i = 0;
     p = findFIRST('\0',argv[1]);
     while(p != (char *)-1) {
       printf("%s\n",decodF(p));
       p = findNEXT();
       ++i;
     }
     if(i == 0) printf("File not found\n");
}
```

Notes:

- Decodf(fcb) decodes a file control block into a printable ASCII file name in the usual CP/M format.
- This function uses the default buffer at BOOT+0x80. In most applications you have to copy the file control block to safety or else use decodF() and copy the return string to safety.
- findNEXT() MUST follow an initial call to findFIRST() without any intervening disk I/O than can disrupt the default buffer.
- On other systems, findnext() is just bdos(18,0) or ccBDOS(18).

The fixfile Function

Purpose:

Inserts drive name, file name and extension defaults into a string that contains a supposed file name. Does not process wildcards.

Function Header:

```
VOID fixfile(filename,pattern)
char *pattern;                  Pattern to be used to fix the
                                file name.
char *filename;                 Name of file, null-terminated.
                                Must be large enough to receive
                                the fix.
```

Returns:

Nothing. Area assigned to the filename is over-written.

Example: Fix a file name entered by the user.

```
#include <stdio.h>
main()
{
char *p,s[129],t[129];
     printf("Enter a file name: ");
     gets(t);
     strcpy(s,t);
     fixfile(s,p="A:");
     printf("pattern = %s, fixed file name = %s\n",p,s);
     strcpy(s,t);
     fixfile(s,p="B:.C");
     printf("pattern = %s, fixed file name = %s\n",p,s);
     strcpy(s,t);
     fixfile(s,p="A:MYFILE.TXT");
     printf("pattern = %s, fixed file name = %s\n",p,s);
}
```

Notes:

- A null file name will cause the pattern to be used as the complete file name - the pattern is the default file name.
- If the pattern contains a drive spec and the user input does not then the file name is fixed: the drive spec is inserted. A similar action is taken for the other two fields: file name and extension. If the pattern contains a blank field, then no fix is done to the user file name.
- To use fixfile to add a default extension of .COM, use the pattern ".COM". In this case, the user is forced to add the drive name and the file name. The extension .COM will be appended by fixfile if the user entered no extension.

The floor Function

Purpose:

Compute float floor, the largest integer less than or equal to the given number. For example, floor(-1.1) = -2 and floor(1.1) = 1.

Function Header:

```
float floor(f)
float f;            Float argument.
```

Returns:

```
g               Where g is a whole number, signed,
                with g <= f and f < g+1.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float f;
float atof();
float floor();
     printf("Enter float f: "); gets(s); f = atof(s);
     printf("floor(%f) = %f\n",f,floor(f));
}
```

Notes:

- No error codes returned. No need to code for errno.
- Floor() does not truncate. For x > 0, truncate, but for x < 0 subtract one and truncate.
- No overflow check.
- No auto conversion to FLOAT.

The fmod Function

Purpose:

Solves for float f in the equation x = k*y + f such that x*f >= 0, fabs(f) < fabs(y), for some long integer k.

Function Header:

```
float fmod(x,y)
float x,y;          Arguments x, y as above.
```

Returns:

f — As computed above, a remainder (not a modulus). For this float library, k = x/y and f = x - k*y.

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,y;
float atof();
float fmod();
     printf("Enter float x: "); gets(s); x = atof(s);
     printf("Enter float y: "); gets(s); y = atof(s);
     printf("fmod(%f,%f) = %f\n",x,y,fmod(x,y));
}
```

Notes:

- No error codes returned. No need to code for errno.
- No overflow check.
- No auto conversion to FLOAT.
- Fmod() works because the float library converts to integer by truncation. If it didn't, then it would be much more difficult to write.

The fnb Function

Purpose:

Skips over non-blank characters to locate the first blank. Stands for Find Next Blank. Blanks are defined by isspace().

Function Header:

```
char *fnb(s)
char *s;                String, null-terminated.
```

Returns:

The address of the null terminator in the string, or the address of the first blank character. Blanks are defined by isspace().

Example: Find and print the first word in a line of text.

```
#include <stdio.h>
main()
{
char buf[129];
char *s,*t,*sob(),*fnb();

     printf("Enter a line of text: ");
     gets(buf); s = sob(buf); t = fnb(s);
     while(s<t) putchar(*s++);
     putchar('\n');
}
```

Notes:

- This function is almost always used in conjunction with sob(), in order to isolate tokens in a line of text.
- High-overhead functions like scanf() can often be avoided in a portable way by using sob() and fnb().
- Both sob() and fnb() return the address of the null delimiter of the string in case the character class checking runs off the end of the string.
- See also the STRINGS PACKAGE for useful string utilities, all in portable C.
- The source code for fnb():

```
char *fnb(s) char *s;
{
  while(*s != EOS && isspace(*s) == FALSE) ++s; return s;
}
```

The fopen_ Function for C/80

Purpose:

Opens a device or buffered file for I/O as a stream using the mode conventions in the Software Toolworks C/80 library.

Function Header:

```
#include <stdio.h>
FILE *fopen_(file,mode)
char *file;             Null-terminated file name.
char *mode;             Mode string. Options:
                        "r"     Read only. Must exist.
                        "w"     Write (or read/write/seek)
                        "u"     Update. Must exist.
                        "rb"    Binary "r"
                        "wb"    Binary "w"
                        "ub"    Binary "u"
```

WARNING: Use UNIX.H to get modes "r+","w+","a","a+". The above function is low-level. It matches usage in C/80 programs that use the Software Toolworks fopen().

Old C/80 code that uses fopen() should compile and link without change. New source code should use the symbol **fopen_**.

Returns:

```
(FILE *)0       Failure.
(FILE *)fp      File pointer on success.
```

Example: Open a file for read only to see if it exists.

```
#include <stdio.h>
main()
{
FILE *fp,*fopen_();

     if(fp = fopen ("TMP","r"))
     puts("File TMP exists");
     else
     puts("File TMP does not exist");
}
```

Notes:

- An fopen_() operation will fail if IOch[] does not have an empty slot. There are 4 devices LST:,CON:,RDR:,PUN: plus MAXCHN-1 files (usually 6). The macro **nfiles** can be set to less than **MAXCHN-1**, but the number of slots is fixed.

- Devices do not require an fopen_(). If you know the file descriptor, then simply read/write to that stream. Streams with descriptors 252,253,254,255 are devices.

- If nfiles = 3 and MAXCHN = 7 and you try to open file #4, then fopen_() has to call sbrk() to get space for the file control block and the file buffer. This can fail. It can also succeed and fragment the heap, because the sfree() function won't free up a file buffer.

- Under Bell Labs Unix, a file descriptor fd is related to the stream fp by fd = fileno(fp).

- The C/80 fopen_() is more limited than the general fopena() and fopenb() functions, which also allow append. Use UNIX.H to access the latter.

- To use more than 6 files, you have to re-compile IOTABLE.CC and put the new IOTABLE.REL into the library or else link in IOTABLE.REL as a separate module.

- In the C/80 library, read() and write() communicate with files only and operate on quanta of 128 bytes. If you use UNIX.H, then all the rules change. See read() and write() for details plus UNIX.H.

- For safety, use only getc() and putc() when operating on devices CON:, LST:, RDR:, PUN:. The resulting code is much more portable and substantially easier to debug.

- A call to fopen_() in mode "w" creates a new file if it does not exist. If it does exist, then it is erased from the disk directory and a new file is created.

- A call to fopen_() in mode "u" opens the file in "r" mode and then changes its flag to "u". So files opened for update must already exist. Generally, this is the mode used for random files using seek(). Such files are read/write.

The fopen, fopena Functions

Purpose:

Unix Ascii File fopen() implemented for C/80.

Function Header:

```
FILE *fopena(name,mode)
char *name;                 Ascii file name, null-terminated
char *mode;                 Mode string, see below
```

Returns:

```
(FILE *)0           Open failed
(FILE *)fp          Open worked, fp=stream pointer
```

Example: Open a file for append, creating it as necessary, ready to write on the end.

```
#include <unix.h>
#include <stdio.h>
main()
{
FILE *fp,*fopen():
  fp = fopen("MYFILE","a+");
  if(fp) {
    fputs("This goes on the end\n",fp);
    fclose(fp);
  }
}
```

WARNING: In UNIX.H we make this definition:

```
#define fopen  fopena
```

Both fopena(), fopenb() are honest functions. See FOPENB.C for details about fopenb().

Notes:

- Use of this function requires seekend() which adds about 300 bytes to the object code size.

- Unix does not support binary modes. The Unix Modes are r,w,a,r+,w+,a+ as follows:

r	open for read, file must exist
w	open for write, any existing file with the same name is truncated
a	open for append, add onto end of an existing file or create a new one.
r+	open for reading and writing starting at the beginning of the file. The file must already exist.
w+	open for reading and writing starting at the beginning of the file. The file is truncated if it already exists, or created if it does not.
a+	open for reading and writing starting at the end of the file. The file is created if it does not already exist.

- For maximum portability, use stub functions fopena and fopenb for Ascii and Binary opens. These functions present a portable interface to most libraries.

- Binary modes are supported by both fopen() and fopena() library functions. However, the mode string method will cause grave portability problems. The following mode access strings will work. However, it is recommended that you use fopenb() to achieve the desired result.

rb	Binary mode for r (existing file, read binary)
rb+	Binary mode for r+ (existing file, update binary)
wb	Binary mode for w (write binary)
wb+	Binary mode for w+ (overwrite binary update)
ab	Binary mode for a (create or binary append)
ab+	Binary mode for a+ (create or append, binary update)
u	Same as r+
ub	Same as rb+

The fopenb Function

Purpose:

Binary File fopen() for C/80. Not a Unix standard function.

Function Header:

```
FILE *fopenb(name,mode)
char *name;                     Null-terminated file name
char *mode;                     Mode string, see fopen().
                                Usually "r", "w", "a".
```

Returns:

0	Open failed
fp	Open worked, fp=stream pointer

Example: Open a binary file for read, count all the carriage returns.

```
#include <unix.h>
#include <stdio.h>
main()
{
int x,n;
FILE *fp,*fopenb();
  fp = fopenb("MYFILE","r");
  n = 0;
  if(fp) {
    while((x = getc(fp)) != EOF) {
      if(x == '\r') ++n;
      putchar(x);
    }
    fclose(fp);
    printf("\n%d carriage returns\n",n);
  }
}
```

WARNING: In UNIX.H the following definition is made:

```
#define fopen   fopena
```

Note that fopena(), fopenb() are honest functions and not macro definitions.

Notes:

- Use of this function requires seekend() and its corresponding overhead of about 300 bytes.
- Unix does not support binary modes. The Unix Modes are r,w,a,r+,w+,a+. See fopena.c.
- For maximum portability, use the functions fopena() and fopenb() for Ascii and Binary opens. These functions present a portable interface to the various modes.
- Binary modes are supported by both fopen() and fopena() library functions. However, mode string usage differs across compiler libraries. To be safe, use fopenb().

The fprintf Function

Purpose:

Outputs formatted data to an open stream using a control string and an appropriate argument list of variable length.

Function Header:

```
fprintf(fp,control,arg1,arg2,...);
FILE *fp;                          open stream
char *control;                     control string, see below
arg1,arg2,...                      appropriate arguments,
                                   8, 16 or 32-bit data, see below
```

Returns:

Nothing.

Example: Print an Ascii chart in Decimal, Octal, Hex, Binary.

```
#include <stdio.h>
main()
{
int i;
  for(i=0;i<128;++i)
     fprintf(stdout,"%-3d  %03o  %02x  %016b\n",i,i,i,i);
}
```

Notes:

- Syntax and usage follows printf() exactly. See printf() for all the tricks.
- In this library, fprintf() is supplied in two versions. The fast version lacks some of the features found in the denser version for longs and floats. Both versions support multiple arguments.
- Fprintf() is not recursive. You cannot do something like fprintf(stdout,"%s",f(s)) where f(s) calls sprintf(). Across libraries, such coding appears to be non-portable.

The fputs Function

Purpose:

Prints a null-terminated string to an open stream. Writes all characters in the string, except the terminating null.

Function Header:

```
int fputs(s,fp)
char *s;                Source string base address
FILE *fp;               Open stream pointer
```

Returns:

```
0                       Operation completely successful.
EOF                     Error occurred.
```

Example:

```
#include <stdio.h>
main()
{
char s[129];
     fprintf(stderr,"Enter a string: ");
     gets(s);
     fputs(s,stderr);
     fputs("\n",stderr);
}
```

Notes:

- A newline is NOT appended, as is the case for puts().
- If the string is void, then no character is output and 0 is returned. If an error occurs or End of Media is detected on the output file, then EOF is returned.
- Unfortunately, the C/80 standard under CP/M is to bail out to the system when an output error occurs. This is because CP/M cannot gracefully handle a full disk situation.
- The standard for fputs() requires that some character other than EOF be returned if the operation succeeds. We chose 0, rather than the last character output by putc(), in order to avoid problems with sign extension.

The fread Function

Purpose:

Transfers bytes from a stream file to main memory. Move bytes in n packets of size s.

Function Header:

```
int  fread(buff,s,n,fp)
char *buff;                    Buffer of data
int  s;                        Size of each data element
int  n;                        Number of elements
FILE *fp;                      Stream pointer, open stream
```

Returns:

n	The number of packets of size s that were actually transferred.
0	Error, including end of file

Example: Read 10 long integers into an array.

```
#include <unix.h>
#include <stdio.h>
main()
{
auto
long nl[10];
FILE *fp,*fopenb():
  fp = fopenb("LONGDATA.DAT","r");
  if(fp) {
     fread(nl,sizeof(long),10,fp);
     fclose(fp);
  }
}
```

Notes:

- Use feof() to distinguish a disk error from end of file.
- Useful for reading in more than 64k without the use of a program loop.
- The primitive read() is used implicitly, but all Unix redirection is in force, because of the explicit use of getc().
- fread() is slower than the primitive read().
- The UNIX.H definition of read() is readа(), which does slow, correct Unix I/O.

The free and cfree Functions

Purpose:

Free() releases memory assigned by malloc(). Cfree() is a synonym often used with calloc().

Function Header:

```
int  free(a)
char *a;                Character pointer to contiguous area
                        obtained from malloc().

int  cfree(a)
char *a;                Character pointer to contiguous area
                        obtained from calloc().
```

Returns:

```
 0   if the request to free the area was honored
-1   if the request failed
```

Example: Get 2048 bytes from malloc(), then free it.

```
#include <unix.h>
#include <stdio.h>
getmem()
{
char *p;

     p = malloc(2048);
     if(p == (char *)0) puts("Request failed");
     else
     free(p);
}
```

Structure used by malloc(): The following 4-byte header appears just before the base address returned by malloc(). It is used by both malloc() and free(), so be careful not to corrupt it.

```
  struct block {
          struct block
              *nxtblk;
          unsigned
                  siz;
  };
```

Notes:

- Code for free() derived from K&R(1978), pp 174-177.
- Brk() and Sfree() don't know about malloc() or free().

The freopen, freopa Functions

Purpose:

Substitutes a new file for an open stream. The name and access mode may be changed during the call.

Function Header:

```
FILE *freopen(name, mode, fp)
FILE *freopa(name, mode, fp)
char *name;                    File name, null-terminated
char *mode;                    Mode "r", "w", "a", see below
FILE *fp;                      Stream pointer, open stream
```

WARNING: In UNIX.H the following definition is made:

```
#define freopen freopa
```

The function freopa() is an Ascii re-open function. It is not supposed to implement binary I/O. The binary re-open function is called freopb().

Example: Re-assign stream stdout to a file at run time without using re-direction on the command line. This example appends a fixed output file each time it runs.

```
#include <unix.h>
#include <stdio.h>
main()
{
extern int fout;
FILE *fp,*freopen():
  fp = freopen("OUTPUT","a",stdout);
  fout = fileno(fp);
  printf("This message should go to file OUTPUT\n");
}
```

Notes:

- Always closes the existing stream before the re-open. This function does not do binary opens. See freopb().
- Descriptors fin and fout are fixed as required.
- It so happens that this particular freopa() can handle 3-char mode strings and hence binary opens. However, for maximum portability, use freopb().
- There is no way to re-assign the descriptors fin and fout to reflect stdin and stdout changes without going through the process in the example. This is one of the very few places where the Unix V7 standard is violated.

The freopb Function

Purpose:

Substitutes a new file for an open stream. The name and access mode may be changed during the call. The new open is done in binary mode.

Function Header:

```
FILE *freopb(name, mode, fp)
char *name;                     File name, null-terminated
char *mode;                     Mode string "r", "w", "a"
FILE *fp;                       Stream pointer, open stream
```

WARNING: In UNIX.H the following definition is made:

```
#define freopa  freopen
```

The functions freopen and freopa are Ascii re-open functions. They cannot implement binary.

Example: Re-assign stream stdin from a command line re-direction argument. This example reopens stdin as a binary file so we can read characters without CR/LF translation. The example counts the number of CR/LF pairs.

```
#include <unix.h>
#include <stdio.h>
main()
{
extern int fin;
extern char *p_fin;
int x,y;
FILE *freopb();
  if(p_fin == (char *)0) exit();
  fin = fileno(freopb(p_fin,"r",stdin));
  y = 0;
  while((x = getchar()) != EOF && x != 26) {
     if(x == '\r' && getchar() == '\n') ++y;
  }
  printf("File %s contains %u lines\n",p_fin,y);
}
```

The descriptor fin and the file name pointer p_fin are special to this library. It is here that we see some real differences between Bell Labs Unix V7 and the present upgrade from Unix V3. However, the use being made is not portable to Unix anyway - binary I/O is not featured.

Notes:

- o Always closes the existing stream before the re-open.
- o File descriptors fin and fout are fixed as required.
- o This function does binary opens. See freopen() for Ascii re-opens.
- o There is no way to re-assign the descriptors fin and fout to reflect stdin and stdout changes without going through the process in the example. This is one of the very few places where the Unix V7 standard is violated. See also freopen, freopa.

The frexp Function

Purpose:

Splits x into x = f*(radix**n) where n is an integer and the value of f satisfies 0.5 <= f < 1.0. The value of radix is 2 for this library.

Function Header:

```
float frexp(x,nptr)
float x;                Float to be split into mantissa &
                        exponent.
int *nptr;              Where to store the integer exponent.
```

Returns:

```
f           The fraction, a FLOAT.
n           via *nptr = n;
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
int n;
float x,f;
extern int errno;
float atof();
float frexp();
  errno = 0;
  printf(
  "frexp(x,nptr) splits float x into mantissa & exponent\n");
  printf("Enter a float x: "); gets(s); x = atof(s);
  f = frexp(x,&n);
  printf("frexp(%f,nptr) = %f, *nptr = %d, errno = %d\n",
         x,f,n,errno);
}
```

Notes:

- The float exponent in this library is r = ((int)255 & (int)x.c[3]) - (int)128, where the float is expressed as a union { char c[4]; float f; } x. The float library is radix 2.

- There is always a better way to do this calculation based on knowledge of the float representation.

The fscanf Function

Purpose:

Parses formatted input text from a specifed open stream.

Function Header:

```
#include <stdio.h>
int fscanf(fp,control,&arg1,&arg2,...)
FILE *fp;                              Open stream pointer
char *control;                         Control string, see below
&arg1,&arg2,...                          Appropriate arguments
                                       See below for rules.
```

Returns:

The number of successfully parsed arguments. An error occurred if the number returned does not match the number of arguments following the control string. **C/80 requires that fscanf() be enclosed in parentheses** in order to check the return value, i.e.,

```
                i = (fscanf(fp,"%d",&x));      RIGHT WAY
rather than
                i = fscanf(fp,"%d",&x);        WRONG WAY
```

Example: The following reads numbers from a file entered on the command line until either a data file error occurs or end of file is reached.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
FILE *fp,*fopen();
int x;
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Open failure"); exit(0);
     }
     while((fscanf(fp,"%d",&x))>0) printf("%d\n",x);
}
```

Notes:

- The scanf family of functions accepts only addresses for its argument list. To error-check coding, verify that each argument has the address operator & as a prefix, or else the argument is a pointer (hence already an address).

- The converted values are stored at the given addresses in order left to right. Skips in the control string do not have an argument so the argument count may not match the conversion count.

- Short counts may hang the run-time package. Overly abundant counts will leave variables unfilled at run time.

- Always check the return of fscanf() to see if matches the expected value. It is easy to program infinite loops using fscanf().

- C/80 and its multiple-argument kludge cause us to write parentheses around fscanf() in order to recover the returned value. For example,

```
if((fscanf(fp,"%d",&x)) > 0)
```

  will not work under C/80 with the extra parentheses removed.

- To turn on float or long libraries for use with fscanf(), use the appropriate switches:

```
#define sfLONG  1       /* turn on long fscanf */
#define sfFLOAT 1       /* turn on float fscanf */
```

  The compiled code will change in size according to how much of the float and long libraries are actually used.

The fseek Function

Purpose:

Positions the read/write pointer in an open file. Has no effect on devices like the console or printer.

Function Header:

```
int  fseek(fp,offset,position)
FILE *fp;                       Stream pointer, open stream
long
offset;                         Long integer offset from position
                                described below. Offset can be
                                negative.
int
position;                       0 = Beginning of file
                                1 = Current position in file
                                2 = End of the file
```

Returns:

```
 0      It worked
-1      It failed
```

Example: Seek to the last record of a CP/M file that was opened in binary mode.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
int x;
FILE *fp,*fopenb();

  if(argc <= 1) exit():
  fp = fopenb(argv[1],"r");
  if(fp) {
     if(fseek(fp,-128L,2) == 0) {
       puts("seek worked");
       while((x = getc(fp)) != EOF) putchar(x);
     }
     else
     puts("Bad seek");
  }
}
```

WARNING: Numbers like 128 are 16 bits whereas 128L is 32 bits. Look out for stack misuse with functions like fseek.

Notes:

- Requires the long/float library in order to be used. This has been automated through the header file STDIO.H. If it doesn't work, then use #define mathlib 1 to get the required library search.

- The largest file possible under CP/M-80 is 65536*256 = 16777216 bytes.

- The second argument of fseek() is a long, 32 bits. The most common usage error is to write in a small number not cast as a long integer. In the example we used -128L. To use -128 instead is a serious error, as it causes the stack to be off by 16 bits. Note that the error would not occur if we used a variable for the second argument or if the number used was larger than 32767.

The C/80 Standard ftell and ftellr Functions

Purpose:

Reports the position of the file pointer for an open stream.

Function Header:

```
int ftell(fp)
FILE *fp;                   Open stream pointer

int ftellr(fp)
FILE *fp;                   Open stream pointer
```

Returns:

ftell: Byte count from the beginning of the file, or for files over 64k bytes, the position of the file pointer in the current sector.

ftellr: Number of bytes from the beginning of the file divided by 256 (number of whole 256-byte sectors).

WARNING: The header file UNIX.H re-defined ftell to be ftellu, which returns a long integer. The above functions are primitives that are used by ftellu. For future compatibility, try to use the long integer return of ftellu (or ftell with UNIX.H).

Example: Print the file pointer position for an open stream.

```
VOID report(fp) FILE *fp;
{
int i; unsigned amt,ftell();

       amt = ftellr(fp);
       if(amt < 256) { /* then ftell gives byte count */
         printf("Pointer at byte %u\n",ftell(fp));
       }
       else {
         printf("Pointer at sector %u, byte %u\n",amt,ftell(fp));
       }
}
```

Notes:

- The Bell Labs Unix V7 compatible library CLIBU.REL uses ftellu() which returns a long integer answer. If you do NOT include the header file UNIX.H, then the standard C/80 ftell will be used.

- If you use UNIX.H, then ftell, ftellr and ftellu will appear in the symbol table from M80/L80. Your code will reference ftell() but actually use ftellu().

The ftell and ftellu Functions

Purpose:

Reports the position of the read/write pointer in an open disk file. Not for devices like the console or printer.

Function Header:

```
#include <unix.h>
long ftell(fp)
FILE *fp;                Stream pointer, open stream
```

Returns:

```
-1      Failure
lp      long integer offset of read/write pointer
        from the beginning of the file.
```

WARNING: In UNIX.H is made the definition

```
#define ftell   ftellu
```

This definition allows you to use the Unix calls with standard long integer arguments. If you want to mix and match, then undo the definition and use ftellu().

Example: Report the position in an open file.

```
#include <unix.h>
#include <stdio.h>
findPosition(fp)
FILE *fp;
{
long ftell();

     printf("Position of file is %lu\n",ftell(fp));
}
```

Notes:

- Requires the long/float library in order to be used. This is automated in STDIO.H
- The largest file possible under CP/M-80 is 65536*256 = 16777216 bytes.
- You must make a declaration for ftell() as in the example. The header files and the library cannot do it for you.
- It is a disaster to use ftell() returns as an integer. The return conventions for integers versus long are assumed to be different. It is an accident if it works at all.

The ftoa Function

Purpose:

Converts a 32-bit internal format floating point number into a null-terminated string of decimal digits. Decimal point, precision and exponent field options exist. SPECIAL C/80 MATHPACK FUNCTION.

Function Header:

```
ftoa(how,pr,f,s)
char how;               Conversion 'E', 'F', 'e', 'f', 'g'
int pr;                 Precision, 6 is K&R default.
float f;                Float to be converted to ASCII.
char *s;                Storage string for digits and exponent.
```

Returns:

Storage s is filled with converted digits and exponent field, null-terminated. This area must be large enough to hold the conversion.

Example: Get a float from the console and print without using the printf() family of functions.

```
#include <stdio.h>
main()
{
float f,atof();
char s[129];

    fputs("Enter a float: ",stderr);
    gets(s);                    /* e.g., s="23e-1" */
    f = atof(s);                /* f is 4 bytes */
    ftoa('f',6,f,s);            /* e.g., s="2.300000" */
    fputs(s,stderr);
}
```

Notes:

- Floats are 32 bits (4 bytes). Storage of a float is documented in the C/80 Mathpack. See also the Transcendental function library for information about the floating point exponent.
- This function differs for every float library implementation. The best way to avoid facing the rules is to use sprintf().
- The source for ftoa() is provided in assembler with The Software Toolworks MathPack.

The fwrite Function

Purpose:

Transfers bytes from main memory to a stream file. Move bytes in n packets of size s.

Function Header:

```
int  fwrite(buff,s,n,fp)
char *buff;                    Buffer for data
int  s;                        Size of each data item
int  n;                        Number of data items
FILE *fp;                      Stream pointer, open stream
```

Returns:

n	The number of packets of size s that were actually transferred.
0	Error, including end of file

Example: Write a buffer of 4096 bytes to an output file. Return number of bytes actually written.

```
#include <unix.h>
#include <stdio.h>
int doWrite(fp,buf)
FILE *fp;
char buf[4096];
{
int x;
     x = fwrite(buf,1,4096,fp);
     return (x);
}
```

WARNING: Some libraries on other target systems foolishly write this function to some unknown standard that returns less than the number of packets requested, even though the write did not fail. They break on terminators like carriage return and line-feed. Beware as you try to transport this particular function.

Notes:

- Useful for writing out more than 32k without the use of a program loop.
- The primitive writea() is used implicitly, but all Unix re-direction is in force, because of the explicit use of putc().

The getatt Function

Purpose:

Get the CP/M file attributes, which are:

	ATTRIBUTE NAME	SYMBOLS USED
1.	Read-only	$R/O or $R/W
2.	System	$SYS or $DIR
3.	Archive	Used by some backup programs, but not by CP/M at present

Function Header:

```
int  getatt(name)       CP/M coded attribute, see below
char *name;             File name, usual CP/M conventions
```

Returns:

```
-1      File not found, otherwise
n       Coded attribute word, 0 <= n <= 7,
        where:

          n&1 = Read-only attribute
          n&2 = System attribute
          n&4 = Archive attribute
```

Example: Print the attributes of a CP/M file entered on the command line.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int n;
    if(argc <= 1) {
        puts("Usage: A>main filename"); exit();
    }
    if((n = getatt(argv[1])) == -1) {
        puts("File not found"); exit();
    }
    printf("Attributes for file %s\n",argv[1]);
    puts( (n&)1 ? "$R/O" : "$R/W");
    puts( (n&2) ? "$SYS" : "$DIR");
    puts( (n&4) ? "Archive set" : "Archive not set");
}
```

Notes:

- The name is a normal C string with NULL delimiter.
- To decode the attributes on return, use the logical AND operator & as outlined in the example above.

The getbyte Function

Purpose:

Reads a byte from the console, no-echo, even though going through the standard C library function getchar() to obtain the character.

Function Header:

```
int getbyte()
```

Returns:

x — The byte read from the keyboard, no translation, or from open stream stdin, in which case translation can occur.

Example: Get a yes or no answer from the user.

```
#include <stdio.h>
main()
{
     while(TRUE) {
       fputs("Do you want to continue? ",stderr);
       if(toupper(getbyte()) != 'Y') break;
       puts("YES");
     }
     puts("NO");
}
```

Notes:

o Below is the source code for getbyte(). It does a direct BIOS call via getchar() to obtain the character without console echo and without any kind of translation. The reason for the BIOS call is to allow NULL to be read as a character (CP/M function 6 disallows NULL).

```
int getbyte()
{
extern char Cmode;
int x,y;
  y = Cmode; Cmode = 2;
  x = getchar();
  Cmode = y;
  return x;
}
```

o The function getbyte() gets characters from files under stdin re-direction. The console I/O will be the same, but character translation will depend upon the open mode of the file. See freopen() for ways to invoke binary mode. See the Binary Flags section and GC_BIN for other ideas.

The getc Function

Purpose:

Reads the next character from an open stream. Not a macro.

Function Header:

```
int getc(fp)
FILE *fp;                Open stream pointer
```

Returns:

```
-1              If end of file was reached. We define EOF to be -1.
 c              Next character from stream fp, on success.
```

Example: The following reads characters from a file entered on the command line until either a data file error occurs or end of file is reached. Each character is printed as it is read.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int x;
FILE *fp,*fopen();
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Open failure"); exit(0);
     }
     while((x = getc(fp)) != EOF) {
          printf("%c",x);
     }
}
```

Notes:

- Expect getc to be a macro in most C libraries. It's not in this one.
- Under CP/M, the BIOS will complain on a disk read error, which is the only possibility besides end of file. In this case, the system will kill the running program.
- A return of -1 under C/80 means that end of file was encountered. Other C systems will require that you check EOF with feof(). The RDR: device never returns EOF, but CON: does in line mode.
- Beware of mixing file descriptors and stream pointers. While C/80 will buy it, other systems won't. The connection is fd = fileno(fp), where fd is an integer and fp is a stream.
- Both CON: and RDR: devices are recognized. The RDR: device does not translate characters, but CON: does unless in binary mode.

The getcbinary Function

Purpose:

Reads the next character from an open stream in binary mode. Not a macro. Recognizes CON: and RDR: devices. Not a K&R standard function.

Function Header:

```
int getcbinary(fp)
FILE *fp;                Open stream pointer
```

Returns:

-1	If end of file was reached. We define EOF to be -1. Disabled for CON: and RDR:.
c	Next character from stream fp, on success, with no translation of any kind.

Example: The following reads characters from stdin in binary mode until EOF or ctrl-Z is encountered. The total carriage return count is reported.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
int x,y;
     y = 0;
     while((x = getcbinary(stdin)) != EOF && x != 26) {
          if(x == '\r') ++y;
     }
     printf("%d returns\n",y);
}
```

Notes:

- Both CON: and RDR: devices are recognized.
- A return of -1 under C/80 means that end of file was encountered. The RDR: device never returns EOF, but CON: does in line mode. In binary mode, CON: returns the characters entered without translation.
- For portability, do not use this function. It is a library internal documented for your convenience. It is not supported on most systems.

The getchar Function

Purpose:

Reads the next character from the standard input stream stdin. Not a macro in this library.

Function Header:

```
int getchar()
```

Returns:

c	Next character from stream stdin, on success.
EOF	If end of file was reached. We define EOF to be -1. The end of file character for the console is ctrl-Z. For files (under re-direction), it is ctrl-Z or EOF.

Example: The following reads characters from the console until end of file is reached. Each character is printed as it is read.

```
#include <stdio.h>
main()
{
int x;
     while((x = getchar()) != EOF) {
          printf("%c",x);
     }
}
```

Notes:

- Expect getchar to be a macro in most C libraries. It has no side effects.
- Under CP/M, the BIOS will complain on a disk read error, which is the only possibility besides end of file. In this case, the system will kill the running program unless you insist upon continuing with the error.
- A return of -1 under C/80 means that end of file was encountered. Other C systems will require that you check EOF with feof().
- A normal use of getchar() may use the companion function ungetc(x,stdin). This function puts the input value x back onto the stream so that the next getchar() call receives x.
- getchar() reads from stream stdin, which may in fact be a file due to re-direction. For console properties see CHmode().
- Binary console mode (Cmode <> 1) turns off editing, ctrl-B processing, CR/LF translation and EOF return for ctrl-Z.

The getddir Function

Purpose:

Fills an array char *p[] with file names and sizes, using a wildcard mask to direct the file selection. Total memory use for 255 files is about 5k bytes. The programmer handles memory management.

Function Header:

```
int getddir(s,p,nmax)
char *s;                    Wildcards for file name.
char *p[];                  p[i] points to 17-byte area.
int nmax;                   Size of array p[].
```

Returns:

n	Number of files processed. n <= nmax.
p[i][0] - p[i][14]	Null-terminated filename in special format "A:FILENAME.EXT".
p[i][15]	Last extent, 0-255.
p[i][16]	Number of records in the last extent.

Example: Print a sorted directory with file sizes for any mixed combination of wildcards across any number of disk drives.

```
#define RECLEN 17
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *w; static int i,j,k,n; static unsigned total;
char s[129],*p[255],q[255*RECLEN];

  for(j=i=0;i<255;++i,j += RECLEN) p[i] = &q[j];
  total = n = 0; i = 1;
  if(argc==1) argv[++argc] = "";
  do {
     strcpy(s,argv[i]);
     if(index(s,'.')==0) strcat(s,"*.*");
     n += getddir(s,&p[n],255-n);
  } while(++i<argc && n<255);
  ssort2(n,p,RECLEN);
  for(j=i=0;i<n;++i) {
       k = kilos(p[i]); total += k;
       printf("%-14s%3dk",p[i],k);
       if(++j==4) {putchar('\n'); j = 0;}
       else putchar(' ');
  }
  if(j) putchar('\n');
  printf("%d Files using %u kilobytes\n",n,total);
}
```

```
kilos(r) char *r;
{
  return (16*r[15]+(7+r[16])/8);
}
```

Notes:

- For CP/M 2.2xx only. Library hooks are documented in STDIO.H.
- To just print the file names, use puts(p[i]). The null terminator always appears before the extent and last record count.
- The file size is usually 128*recs+16384*extent. This will require long integer arithmetic in general. Ascii files mark the end with ctrl-Z (usually in the last record of the file).
- To get the file size in kilobytes, using only integer arithmetic, use 16*extent + (recs+7)/8.
- Extents of a file need not be consecutively located in the directory tracks. Further, two extents may be lumped into one entry. A full directory entry uses 16k bytes.
- The complete source for the function getddir follows. It is often the case that this program is engineered for a special application. Use the archives to get a working copy.

```
int getddir(s,p,nmax)
char *s;                    /* wildcards for file name */
char *p[];                  /* p[i] points to 17-byte area */
int nmax;                   /* size of array p[] */
{
static char *q;
static int i,n,drive;
char *findFIRST(),*findNEXT(),*decodF();

  if((q = findFIRST('?',s)) == (char *)-1) return 0;
  drive = (s[1]==':') ? ((s[0]|32)-'a') : (logged());
  n = 0;
  do {
       if(n >= nmax) break;
       q[0] = drive+1;
       strcpy(p[n],decodF(q));
       for(i=0;i<n;++i){
         if(strcmp(p[i],p[n])==0 && p[i][15]<q[12]) break;
       }
       p[i][15] = q[12];                    /* extent */
       p[i][16] = (255 & q[15]);            /* records */
       if(i == n) ++n;
  } while((q = findNEXT()) != (char *)-1);
  return n;
}
```

Extended Directory Utility

Below is the source code for a sorted directory utility which is an extension of the example above. Source appears in the archives.

The features:

- o Displays sorted directory on any number of drives for any number of wildcard filespecs. Output to disk or printer via I/O re-direction.
- o Gives disk space report on command which includes group size and disk capacity.
- o Output can be in 1,2,3,4 columns.

Command strings can be as long as the CP/M command line. Here are some examples:

```
A>dd A: B: C:          Disk directories of drives A:,B:,C:
A>dd A:                4-column directory of drive A:
A>dd A: ;1             1-column directory of drive A:
A>dd A: ;2             1-column directory of drive A:
A>dd A: ;3             1-column directory of drive A:
A>dd A: ;              Directory of drive A: plus free space.
A>dd A:*.COM           Display all files ending with .COM
A>dd .com              Same as A:*.COM
A>dd A:f*.*            Display all files starting with f.
A>dd f                 Same as A:f*.*
A>dd ?.                Display one-letter file names.
A>dd B:*.com ;         Display .COM files on B: plus free space.
A>dd B:*.C *.COM       All .C files and .COM files on B:.
```

A selected drive remains the default, as in the last example above. Free space reports are issued when encountered, therefore they will occur before file displays. A disk free space report causes a drive reset, in order to insure that disk changes are recognized.

```
#include <stdio.h>
kilos(r) char *r;
{
    return (16*r[15]+(7+r[16])/8);
}

struct dfree {
    unsigned avail;     /* free groups */
    unsigned total;     /* total groups */
    unsigned bpsec;     /* bytes per sector */
    unsigned spg;       /* sectors per group */
};
```

```
freespace(d)
int d;
{
char s[3];
unsigned kbytes; struct dfree df; unsigned getdfree();

    if(d==0) d = logged(); else --d;
    s[0] = d+'A'; s[1] = ':'; s[2] = 0;
    kbytes = getdfree(s,&df);
    printf("%c: %uk free in %d groups\n",
           s[0],kbytes,df.avail);
    printf("Disk capacity %dk using %d-byte groups\n",
           df.total*(df.bpsec*df.spg/1024),
           df.bpsec*df.spg);
}

#define RECLEN 17
main(argc,argv) int argc; char **argv;
{
static int i,j,k,n,w,drive; static unsigned total; int getddir();
char fcb[36],s[129],*p[255],q[255*RECLEN];

    for(j=i=0;i<255;++i,j += RECLEN) p[i] = &q[j];
    total = n = 0; i = 1; w = 4;
    if(argc==1) argv[++argc] = "";
    drive = logged();
    do {
        strcpy(s,argv[i]);
        if(index(s,'.')==0) strcat(s,"*.*");
        makeFCB(fcb,s);
        if(fcb[0]) drive = fcb[0]-1; else fcb[0] = drive+1;
        if(fcb[1]==';') {
            w = fcb[2]-'0'; if(1<=w && w<=4) continue;
            if(w == '?'-'0') { freespace(fcb[0]); continue;}
            w = 4;
        }
        if(fcb[1]==' ') fcb[1]='*';
        strcpy(s,decodF(fcb));
        n += getddir(s,&p[n],255-n);
    } while(++i<argc && n<255);
    ssort2(n,p,RECLEN);
    for(j=i=0;i<n;++i) {
        k = kilos(p[i]); total += k;
        printf("%-14s%3dk",p[i],k);
        if(++j >= w) {putchar('\n'); j = 0;}
        else putchar(' ');
    }
    if(j) putchar('\n');
    if(n) printf("%d Files using %u kilobytes\n",n,total);
}
```

The getdfree Function

Purpose:

Computes the number of free kilobytes on the disk. Return disk drive info. See below. Disk drive is entered as a string.

Function Header:

```
unsigned getdfree(drive,dfp)
char *drive;                    Drive, e.g., "A:"
struct dfree *dfp;              See the example below.
```

Returns:

u Free disk space in kilobytes.

\+ Additional information is returned in the 4-word structure passed as the second function argument. The array is organized as follows:

DESCRIPTION	ARRAY	STRUCTURE
Free group count	info[0]	dfp->avail
Full disk group count	info[1]	dfp->total
Bytes per logical sector	info[2] = 128	dfp->bpsec
Logical sectors per group	info[3] = 8 or 16	dfp->spg

Example: Print disk drive statistics for any drive.

```
struct dfree {
   unsigned avail;          /* free groups */
   unsigned total;          /* total groups */
   unsigned bpsec;          /* bytes per sector */
   unsigned spg;            /* sectors per group */
};
#include <stdio.h>
main()
{
unsigned kbytes;
char s[129];
struct dfree df;
unsigned getdfree();
   fputs("Enter disk drive letter: ",stderr);
   gets(s);
   kbytes = getdfree(s,&df);
   printf("%uk free on %c:\n",kbytes,s[0]);
   printf("%d free groups (group size = %d bytes)\n",
           df.avail, df.bpsec*df.spg);
   printf("Disk capacity %d groups\n",df.total);
}
```

Notes:

- For CP/M 2.2xx only. Library hooks are documented in STDIO.H.
- The current version resets the disk system each time it is called. This feature logs in drive A: and also the drive to be accessed. All other disks are reset.
- The disk free space report has the additional feature of taking proper action for diskette changes. Reports the correct free space regardless of the number of disk swaps.
- Bad drive specs are mapped to drive A:. Some bad drive input can make its way to the program, in which case CP/M itself will abort.
- The complete source for the program getdfree follows. It is often the case that this program is altered and engineered for a particular application. See the archives to get a working copy. Below is for reference during brainstorming.

```
#define CURDISK         25          /* BDOS                 */
#define RESETDRIVES     13          /* BDOS                 */
#define LOGDISK         14          /* BDOS                 */
#define SELDSK          9           /* BIOS                 */
#define SECSIZ          128         /* Logical sector size  */

struct dfree {
        unsigned avail;         /* free groups       */
        unsigned total;         /* total groups      */
        unsigned bpsec;         /* bytes per sector  */
        unsigned spg;           /* sectors per group */
};

unsigned getdfree(drive,dfreep)
char *drive;
struct dfree *dfreep;
{
/* CP/M 2.x Disk Parameter Tables */

struct dpb {            /* Disk Parameter Block */
        unsigned spt;   /* Sectors/track */
        char bsh;       /* block shift (3 = 1k, 4 = 2k, 5 = 4k) */
        char blm;       /* a bit mask 'bsh' bits wide */
        char exm;       /* extent mask  */
        unsigned dsm;   /* maximum block number for drive */
        unsigned drm;   /* maximum directory entry index */
        char al[2];     /* Directory Allocation Bitmap */
        unsigned cks;   /* size of checksum vector */
        unsigned off;   /* Number of reserved tracks */
        } ;             /* pointer obtained from dphp->hdpbp */
```

```
struct dph {            /* Disk Parameter Header */
        char *xltp;     /* pointer to sector translation table */
        int dscr[3];    /* BDOS scratch */
        char *dirbp;    /* Pointer to directory buffer */
        struct dpb *hdpbp; /* Disk Parameter Block pointer */
        char *csvp;     /* Pointer to checksum vector */
        char *alvp;     /* Pointer to Allocation Bitmap */
        } ;             /* pointer returned by bios seldsk */

static
int maxblock,blk,freeblk,disk,cdisk,n;
static
char bitnumber[] = {
        128,64,32,16,8,4,2,1};
static
unsigned *u;
static
char *map;
static
struct dph *dphp;
static
struct dpb *dpbp;

    disk = ccBDOS(CURDISK);
    ccBDOS(RESETDRIVES);
    n = toupper(drive[0]) - 'A';
    cdisk = (0 <= n && n <= 15 && drive[1] == ':') ?
        n : disk;
    ccBDOS(cdisk,LOGDISK);
    dphp = ccBIOS(1,cdisk,SELDSK); /* pointer disk param header */
    dpbp = dphp->hdpbp;            /* pointer disk param block */
    maxblock = dpbp->dsm;          /* max block number, 0=first */
    map = dphp->alvp;              /* pointer to allocation bitmap */
    for(freeblk=blk=0;blk<=maxblock;++blk)
      if((map[blk/8] & bitnumber[blk%8])==0) ++freeblk;

    u = (unsigned *)dfreep;
    *u++ = /* dfreep->avail */ freeblk;
    *u++ = /* dfreep->total */ maxblock+1;
    *u++ = /* dfreep->bpsec */ SECSIZ;
    *u   = /* dfreep->spg   */ (SECSIZ << dpbp->bsh)/SECSIZ;

    ccBDOS(disk,LOGDISK);
    return ( (unsigned)((*u/(1024/SECSIZ))*freeblk) );
}
```

The getl Function

Purpose:

Reads a 32-bit long integer from the stream, in Intel Reverse Format.

Function Header:

```
long getl(fp)
FILE *fp;               Stream pointer, open stream
```

Returns:

```
-1      Error or end of file
 n      32-bit word, otherwise
```

Example:

```
VOID getlongs(fp)
FILE *fp;
{
long x,getl();
int feof();
  while(1) {
    if((x = getl(fp)) == EOF) {
        if(feof(fp) == EOF) break;
    }
    printf("Long x = %ld\n",x);
  }
}
```

Notes:

- To detect end of file, use feof() every time getl() returns a value of -1. Do not check end of file on every read - the overhead of feof() is too much.
- Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. In addition, poorly constructed libraries may write getl() from getc(), which introduces the possibility of CR/LF translation.
- Getl() calls the library primitive getcbinary() to avoid any character translation. Most Unix Ports to newer machines do not translate characters in files anyway. Code that uses getl() on an ASCII file should run without changes on other targets.

The getline Function

Purpose:

Reads a line of text from the console, or the re-direction file for stream stdin, up to and including the newline. The string is null-delimited.

Function Header:

```
int getline(s,n)
char *s;                    Base address of string buffer.
int n;                      Size of the buffer in bytes minus 1.
```

Returns:

EOF	If end of file was read before any other characters.
n	The number of characters read, which always includes a newline character for a successful read operation.

Example: Read a line of text from the console, print it.

```
#include <stdio.h>
main()
{
char buf[129];
int i;
     printf("Enter a line of text: ");
     i = getline(buf,128);
     if(i != EOF) {
       printf("Characters = %d\n",i);
       printf("Data = %s",buf);
     }
}
```

Notes:

- Reading more than the allowed number of bytes without an intervening newline causes the excess characters to be lost.
- Empty input returns 1, not 0, and buf[0] = '\n'.
- At end of file, getline returns -1.

The getln Function

Purpose:

Reads a line of text from the console, or the re-direction file for stream stdin, up to the newline. The string is null-delimited. The newline is stripped from the buffer.

Function Header:

```
int getln(s,n)
char *s;                    Base address of string buffer.
int n;                      Size of the buffer in bytes minus 1.
```

Returns:

EOF	If end of file was read before any other characters.
n	The number of characters read, which always includes a newline character for a successful read operation.

Example: Read a line of text from the console, print it.

```
#include <stdio.h>
main()
{
char buf[129];
int i;
     fprintf(stderr,"Enter a line of text: ");
     i = getln(buf,128);
     if(i != EOF) {
       fprintf(stderr,"Characters = %d\n",i);
       fprintf(stderr,"Data = %s",buf);
     }
}
```

Example: Read lines of text from stream stdin, write to stdout.

```
#include <stdio.h>
main()
{
char buf[129];
     while(getln(buf,128) != EOF) puts(buf);
}
```

Notes:

- Reading more than the allowed number of bytes without an intervening newline causes the excess characters to be lost.
- Empty input returns 1, not 0, and buf[0] = '\0'. Return is the same as getline().
- Overcomes the limitation of getline(), which is the extra newline.
- Replaces the more dangerous and limited gets(), which can write over the data area due to long lines. But for up to 128 bytes of input, gets() and getln() are the same function.
- This function is not in K&R nor is it in the Unix System III standard. It should be. Range and error checking are essential to the production of bullet-proof user interfaces.
- Entry of ctrl-z returns 0. See K&R.

The getpass Function

Purpose:

Reads in a password without console echo.

Function Header:

```
char *getpass(q)        Returns a static storage location

char *q;                Prompt string to issue to console
                        before password is typed. Must be
                        null-terminated.
```

Returns:

Null-terminated password string pointer
to a static area of memory where the user
input resides. This area is 8 bytes long
plus one NULL. It is overwritten on each call.

Notes:

- The standard ccBIOS(3) interface is used so that we read directly from CP/M. This prevents type-aheads.
- The caller must map case and worry about things like delete and backspace.
- The user's carriage return is echoed, and it happens automatically if the character limit is exceeded.

Example: Ask for user password "SPECIAL" and return 0 on success, -1 on failure.

```
VOID getmypassword()
{
#define MYPASSWORD "SPECIAL"
char *p,*getpass();
  p = getpass("Enter password: ");
  cvupper(p);
  if(strcmp(p,MYPASSWORD) == 0) return 0;
  return -1;
}
```

The getpid Function

Purpose:

Get the process ID of the current process.

Function Header:

```
int  getpid()
```

Returns:

Bogus value of 0, for Unix compatibility.

Notes:

o May not make a Unix program work, but it helps it to compile without errors.

The gets Function

Purpose:

Reads characters from open stream stdin and null-terminates the string. Breaks on newline or EOF. The newline is purged from the buffer.

Function Header:

```
char *gets(s)
char *s;                Base address of the storage area.
```

Returns:

```
(char *)0               End of file and string empty.
(char *)0               Stream error and partially filled string.
(char *)s               Success, base of storage area.
```

Example: Read lines from stdin and print until end of file. Note the stripped linefeed and the extra newline on output.

```
#include <stdio.h>
main()
{
char s[129];
     while(TRUE) {
          fprintf(stderr,"Enter a string or ctrl-Z to exit: ");
          if(gets(s) == (char *)0) break;
          fprintf(stderr,"%s\n",s);
     }
}
```

Notes:

- It is impossible to tell whether a null pointer return stands for an error or end of file.
- The array s[] may be filled with invalid data in case of an error.
- Newlines read from the stream are NOT appended to the array s[]. This is different from fgets(), which appends the newline on input. In practise, it means puts() will output exactly the input received by gets().
- There is no way to protect the storage area s[] from long lines when using gets(). It is best to use fgets() in the cases where over-run is possible.
- Under C/80 and this library, gets() is a call to fgets() with buffer size 128. The newline is stripped off upon return from fgets(). This affords users some protection against crashes at the expense of a limited buffer size.

The getw Function

Purpose:

Reads a 16-bit word from the stream, in Intel Reverse Format.

Function Header:

```
unsigned getw(fp)          Unsigned integer, 16 bits
FILE *fp;                  Stream pointer, open stream
```

Returns:

```
-1      Error or end of file
 n      16-bit word, otherwise
```

Example: Read and print integers taken from an open binary file.

```
VOID getwords(fp)
FILE *fp;
{
int x,getl();
int feof();
  while(1) {
    if((x = getl(fp)) == EOF) {
        if(feof(fp) == EOF) break;
    }
    printf("Word x = %u\n",x);
}
```

Notes:

- To detect end of file, use feof() every time getw() returns a value of -1. Do not check end of file on every read - the overhead of feof() is too much.
- Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. In addition, poorly constructed libraries may write getw() from getc(), which introduces the possibility of CR/LF translation.
- Getw() calls the library primitive getcbinary() to avoid any character translation. Most Unix Ports to newer machines do not translate characters in files anyway. Code that uses getw() on an ASCII file should run without changes on other targets. Beware of size restraints: getw() is 16 bits, but on 36-bit machines it may actually be 18 bits (9 bits per byte instead of 8).

The heapsort Function

Purpose:

Heapsort for an array of character strings with easy-to-change comparison method. Changes pointers but not actual data in the string.

Function Header:

```
VOID heaps(base,n,cmp)
char *base[];          Base address of pointer array
int  n;                Number of array elements to sort
int  (*cmp)();         Compare routine for two strings p,q
                       that returns an integer with the
                       same rules used by Unix strcmp(p,q):

                          = 0     strings p,q are equal
                          < 0     p < q
                          > 0     p > q
```

Returns:

Nothing of value

Example: Sort an array of strings.

```
char *p[10] = {"a","C","d","0","9","2","1","3","8","5"};
char *q[10] = {"a","C","d","0","9","2","1","3","8","5"};
int strcmp();
#include <unix.h>
#include <stdio.h>
main()
{
int i;
        heaps(p,10,strcmp);
        for(i=0;i<10;++i) printf("%d. %s, %s\n",i,p[i],q[i]);
}
```

Results from the above program

```
0. 0, a             Numerals are less than letters
1. 1, C             in the ASCII standard
2. 2, d
3. 3, 0
4. 5, 9
5. 8, 2
6. 9, 1
7. C, 3             UPPERCASE is less than lowercase
8. a, 8             in the ASCII standard
9. d, 5
```

Notes:

- You can most often use strcmp() as the argument for cmp().
- It is necessary to declare the function cmp() before the call to heaps(). The required ASM code for the call is LXI H,cmp ! PUSH H. Look out for the incorrect LHLD cmp ! PUSH H.
- To make a compare for mixed upper and lower case string data, write a function called mycmp() and use it instead of strcmp() in the example below.
- This source file is very portable. Use it on any system with minimal C compiler.

Reference:

A C Reference Manual, pp 62-63,
by Harbison & Steele
Prentice-Hall, Englewood Cliffs (1984)

The highmem Function

Purpose:

Computes the highest memory address that is assignable by sbrk().

Function Header:

```
char *highmem()
```

Returns:

The address below the stack which is the last possible address assignable by sbrk(). This implementation assumes 600 bytes of stack space, which is used in the computation in order to give a conservative estimate.

Example: Find out how many bytes of free contiguous memory are available. Do it three ways.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char *highmem(),*lowmem();
unsigned amt,memsize();
     amt = highmem() - lowmem();
     printf("%u bytes available for use by sbrk()\n",amt);
     amt = ((char *)(&argc) - (char *)sbrk(0));
     printf("%u bytes available for use by sbrk()\n",amt);
     amt = memsize();
     printf("%u bytes available for use by sbrk()\n",amt);
}
```

Notes:

- The 600 bytes of slop is a guess. The right amount depends on your program.
- The value returned from highmem() depends on its location in the calling program. If called from main(), then the answer is accurate. If buried deeply in subroutine calls, the answer could be very optimistic.
- The method used for finding out about heap space **works on 8080 machines** with CP/M 2.2. It **will not work with CP/M 3.0. It will not generalize to mainframes or 8086-type CPU machines.** It does **work on a Motorola 68000 with CP/M-68K.**

The Horner Function

Purpose:

Compute a polynomial value by Horner's method..

Function Header:

```
float Horner(x, p, n)
float x;                Polynomial variable x value.
float p[];              Coefficients of polynomial.
int n;                  Degree of the polynomial.
```

Returns:

The computed float value obtained by putting x into the polynomial equation.

Example: Given p[2] = 3, p[1] = 4, p[0] = 6, then 3*(x**2) + 4*x + 6 = x(x(3) + 4) + 6 = x(x(p[2]) + p[1]) + p[0]. The degree of the polynomial is 2 and the factorization scheme is known as Horner's method.

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
static float p[] = {
     6,4,3
};
float atof();
float Horner();
     printf("Enter a float: "); gets(s); f = atof(s);
     printf("For x = %f, 3*(x**2) + 4*x + 6 = %f\n",
             x,Horner(x,p,2));
}
```

Notes:

- No error checking. This function uses the standard factorization method taught in college algebra.
- To understand the method, study the above example carefully. Note in particular the dimension of the array p[] versus the degree of the polynomial generated from the coefficient array.

The index Function

Purpose:

Finds the character position of a byte inside a string.

Function Header:

```
char *index(str,c)
char *str;                    Source string.
char c;                       Character to locate.
```

Returns:

```
&str[i]                       Where c == str[i], on success.
(char *)0                     Failure.
```

Example: Find the colon in a file spec, strip off the file name, print.

```
#include <stdio.h>
main()
{
char s[129],t[129],*p;
     printf("Enter a file name: ");
     gets(s);
     cvupper(s);
     if(p = index(s,':')) strcpy(t,p+1);
     else strcpy(t,s);
     printf("Stripped file name: %s\n",t);
}
```

Notes:

- Usage of this function differs among libraries. The Unix V7 standard, according to Bourne, is the above.
- Index() should be capable of locating the null terminator in a string. For portability, we recommend using p + strlen(p) as a pointer to the end of a string.
- Index() is the same function as **strchr()**. See Harbison & Steele.

The inport Function

Purpose:

Reads a value from a given CPU port.

Function Header:

```
int inport(port)
int port;                       Port number.
```

Returns:

The value read from the port. Assumes data was available. See system notes below and the example.

Example: Read characters from the modem port.

```
#define MODEM      0330   /* Z90A computer, Z80 CPU, 255 ports */
#define OFFSET     0005   /* 8250 UART, line status register */
#define READY      0001   /* test data ready at port         */
#include <stdio.h>
main()
{
     while(TRUE) {
          while((inport(MODEM+OFFSET)&READY)==0 &&
                 bdos(11,0)==0) ; /* wait for keyboard or modem */
          outport(MODEM,x);                  /* ship the byte   */
     }
}
```

Notes:

- On most machines the status of the port must be interrogated in order to read from the data port. Buffered ports may require a different method, namely checking the character count in the buffer.

- Disk I/O ports are often interrupt driven. Reading from the port will require set-up of the interrupt vector for return.

- Clock and timer chips like the 8253 Intel make for good examples of port usage in C programs. See also Analog-to-Digital boards and parallel port interfaces. These appear on music boards and game interfaces. The signals are joysticks, mice, light pens, plotters, graphics tablets and the like.

- Port operations are inherently machine-dependent. Don't expect to write portable code. But do expect READABLE code. Watch out for 8-bit and 16-bit ports on other machines. Not all ports are 8-bits wide.

The insert Function

Purpose:

Inserts a CP/M command tail into the two default file control blocks and then copies the command tail to the CP/M default buffer.

Function Header:

```
VOID insert(tail)
char *tail;                CP/M command tail to be inserted.
```

Returns:

Nothing. Similar to the insert command in the DDT debugger.

Example: Swap in a new command tail into the default buffers prior to chaining to a new overlay. The link program LEXI.COM takes a command line with three file name arguments.

```
#include <stdio.h>
main()
{
extern char *Cbuf;        /* C console buffer is 136 bytes */

     insert(" TEST1.TXT TEST2.TXT OUTPUT.TXT");
     makeFCB(Cbuf,"LEXI");
     chain(0x100,Cbuf);
     puts("LEXI not found");
}
```

Notes:

- Multiple arguments may be used. The command line limit of 128 characters must be obeyed, however.
- All characters are acceptable, however uppercase translation is performed to simulate CP/M handling of command lines.
- The string may contain whatever you wish.
- CP/M 2.2 might demand in certain applications that the information found in the default buffers also be present in the CCP buffers. This is highly unusual, but may be a problem.
- The file control blocks are built at BOOT+0x5C and BOOT+0x6C. Tokens for these file control blocks are the first two found in the command tail.

The instr Function

Purpose:

Computes the offset position for a substring match.

Function Header:

```
int instr(n,str1,str2)
int n;                  Character position to start.
char *str1,*str2;       Source strings, null-terminated.
```

Returns:

```
0           No substring match
k           Substring of str1 matched str2 at position k.
```

Example: Find the word SAILOR inside a sentence typed by the user.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int k;
     if(argc <= 1) exit();
     if(k = instr(1,argv[1],"SAILOR"))
          printf("Word SAILOR found at position %d\n",k);
     else
          printf("Word SAILOR not found in string\n");
}
```

Notes:

- Instr() is similar to the BASIC function of the same name. It takes three arguments. You may not drop arguments (as is done in BASIC).
- The value of instr(1,"abcd","abc") is 1, not zero.
- The value of instr(1,"xabcdabcd","abc") is 2 whereas the value of instr(3,"xabcdabcd","abc") is 6. Counting of the offset index starts at 1. The maximum offset is the string length of str1.
- This function also works with unsigned integers.

The IOTABLE module

Purpose:

Storage area for I/O information, channel data. Used by file routines such as read() and write(), seek(), getc(), putc(), fopen(), fclose(), fflush(), ffb(), fseek(), ftell().

VARIABLE LIST

MAXCHN — Equals 1+n, where n is the maximum number of disk files. Settable by re-assembly to any number from 1 to 251. Currently, n = 6 and MAXCHN = 7.

DEVICES — Numbers 252,253,254,255 are reserved for CON:, RDR:, PUN:, LST: respectively. Number 0 is translated by the software into the console number 252.

`char *IObuf[MAXCHN];`	Addresses of the file buffers
`int IOsect[MAXCHN];`	Sector count in 128-byte hunks
`char IOrw[MAXCHN];`	0 or 1 for last operation, read or write
`int IOtmp;`	Storage for buffer address, work variable
`char IOch[MAXCHN];`	Channel numbers (-1 means empty).
`int IOind[MAXCHN];`	Offset into file buffer for next char
`char IOmode[MAXCHN];`	Mode byte 'r', 'w', 'u'
`char IObin[MAXCHN];`	Binary flag '\0' or 'b'
`char *IOfcb[MAXCHN];`	File control block addresses
`char IOpchan[4];`	Physical device numbers 252,253,254,255
`char IOpread[4];`	Physical read CP/M function numbers 1,3,0,0
`char IOpwrit[4];`	Physical write CP/M function numbers 2,0,4,5
`char IOpeof[4];`	Physical device EOF characters 26,26,26,12
`char *IOdev;`	Physical device string 'con:rdr:pun:lst:' with null terminator, all lowercase.
`int IOnchx[MAXCHN];`	Amount in bytes read by ffb() into buffer
`int IOend[MAXCHN];`	Saves EOF record number for seek
`int IOsize[MAXCHN];`	IO buffer sizes (0 for chan 0, 256 default)

The IOTABLE module

The first **WARNING** to be issued is that offset 0 is not used in any of the arrays. All descriptors start at offset 1 or higher but never exceed MAXCHN-1.

The functions of the I/O library access the IOTABLE, and use the module END.CC, which contains the end-of-program label. This module is pure data (no code segment). It holds the secret to understanding the method by which dynamic buffers are handled in the C library.

Pre-defined file buffers appear in memory as though they were assigned by sbrk(), which means they are located immediately after the physical end of the program. Walt Bilofsky's library puts them into high memory, which is unsuitable for buffer re-allocation and recovery of buffer space via sfree(). The placement of the file buffers in this library allows for programmable buffer sizes and buffer locations.

File buffer size defaults to the size pre-set in STDIO.H. We recommend a buffer size of 256 for C/80 library compatibility. To obtain Unix compatibility, use default buffer size 512.

The size of a file buffer is set by the symbols $SIZ1, ... , $SIZ6. These are assembly language equates that are recognized by IOTABLE. The number of symbols equals the number of possible open files, MAXCHN-1. In order to change the number of files beyond 6, you must edit IOTABLE.CC, re-assemble and invoke LIB.COM to insert the new module into the library.

The macro variable nfiles defined in STDIO.H allows you to select the number of files from 1 to MAXCHN-1. This setting has an effect on the amount of heap space.

The macro variables bufsz1, bufsz2, bufsz3 defined in STDIO.H are used for the purpose of selecting the buffer sizes. The interface allows file buffer size definition directly from language C, in a convenient fashion that is self-documenting.

File control blocks are assigned as needed as though sbrk() were called. Each file control block consumes 36 bytes. File control blocks with IOfcb[fd]=0 will force a call to sbrk() to get space. The construction of file control blocks is handled through routines in XFCB.C, which supports wildcard expansion and uppercase translation, plus white space skipping.

The IOTABLE module

The essential arrays in IOTABLE.CC are IOfcb[], IObuf[], IOsize[]. The address of the file control block is saved in IOfcb[fd]. If the contents of IOfcb[fd] is zero, then fopen() will call sbrk() for 36 bytes of space and update IOfcb[fd] accordingly. Similarly, fopen() checks IObuf[] for a nonzero value, and calls sbrk() for IOsize[fd] bytes as needed, with IObuf[fd] updated to reflect the new buffer address.

To fool the system, you must change IOsize[fd] at a time when the file buffer is empty. The safest method is to rewind the file or change when the file is open and not yet accessed. You can always fool with the buffers and sizes when the channel is unused, but there is no guarantee that an fopen() call will return that channel.

Changes to IOsize[] must be matched with a change of IObuf[]. If the stream is unused, then it is safe to set IObuf[fd]=IOfcb[fd]=0 and call sfree() to free up the buffer space for re-use. A later call to fopen() will automatically get the desired space from sbrk().

As you may have guessed, all the file buffer and file control block space can be recovered by closing all the files and setting IOfcb[] = IObuf[] = 0 for all MAXCHN-1 channels. Use sfree(-1) to get all of memory back. Beware: this invalidates all previous core() and sbrk() function calls!.

The isatty Function

Purpose:

Tests a file descriptor fd to see if it is attached to the console device.

Function Header:

```
int  isatty(fd)
int  fd;          File descriptor fd (as distinguished from a
                  stream pointer fp). Use fileno() to obtain fd
                  from a stream pointer:

                    fd = fileno(fp);
```

Returns:

```
0       fd not attached to console
1       fd is mapped to the console device
```

Example: Test re-direction channel stdin to see if it is set to the console.

```
#include <unix.h>
#include <stdio.h>
VOID whichone()
{
int fd;

    fd = fileno(stdin);
    if(isatty(fd)) puts("Input is from the console");
}
```

Notes:

- The library uses fd = 252 for the console.
- However, 0 is mapped to the console too. For compatibility with defaults, the special value fd = 0 returns isatty = 1.

The isgraph Function

Purpose:

Test for a printable character in the range 041 to 0176 (octal).

Function Header:

```
int  isgraph(c)
char c;                 Character to test
```

Returns:

```
    0       Not a graphic character
NONZERO     Is a graphics character
```

Source code: This function is so often ill-defined and wrongly used that we include the source code to communicate the precise difference between this function and its popular aliases.

```
int isgraph(c)
char c;
{
     if(c == ' ') return 0;
     return isprint(c);
}
```

Notes:

- This range includes all characters from isprint() except SPACE itself.

The istype Functions

Purpose:

The purpose of this group of tools is to classify and test individual characters for inclusion in a particular class.

Function Header:

```
int isascii(c) - Tests c for the range 0 to 127 decimal.
int isalnum(c) - Tests c for alphabetic or numeric.
int isalpha(c) - Tests c for alphabetic.
int isupper(c) - Returns -1 for uppercase c, else 0.
int islower(c) - Returns -1 for 'a' <= c <= 'z', else 0.
int isxdigit(c) - Tests for a valid hexadecimal digit, 0-9,A-F.
int isdigit(c) - Tests for a valid digit, 0-9.
int isspace(c) - Tests for blank, tab, carriage return, linefeed.
int iscntrl(c) - Tests for a control character 0-31 decimal.
int isprint(c) - Tests for a printable character.
int ispunct(c) - Tests for punctuation (not ctrl or alphanumeric).
```

Throughout, c is assumed to be a character, however it may be an integer variable instead. These routines ignore sign extension.

Returns:

```
-1  -  TRUE              The K&R standard is NONZERO return.
 0  -  FALSE
```

Example: Access istype functions by the array method.

```
#include <unix.h>
#include <ctype.h>
#include <stdio.h>
main()
{
/* same example as below, but uses macros and the array method */
}
```

Example: Access standard library functions.

```
#include <stdio.h>
main()
{
int c;
char buf[129];
   while(TRUE) {
     printf("Enter a character: ");
     gets(buf);
     c = buf[0];
     printf( isascii(c) ?
             "isascii(c) = TRUE" : "isascii(c) = FALSE");
     printf( isalnum(c) ?
             "isalnum(c) = TRUE" : "isalnum(c) = FALSE");
     printf( isalpha(c) ?
             "isalpha(c) = TRUE" : "isalpha(c) = FALSE");
     printf( isupper(c) ?
             "isupper(c) = TRUE" : "isupper(c) = FALSE");
     printf( islower(c) ?
             "islower(c) = TRUE" : "islower(c) = FALSE");
     printf( isxdigit(c) ?
             "isxdigit(c) = TRUE" : "isxdigit(c) = FALSE");
     printf( isdigit(c) ?
             "isdigit(c) = TRUE" : "isdigit(c) = FALSE");
     printf( isspace(c) ?
             "isspace(c) = TRUE" : "isspace(c) = FALSE");
     printf( iscntrl(c) ?
             "iscntrl(c) = TRUE" : "iscntrl(c) = FALSE");
     printf( isprint(c) ?
             "isprint(c) = TRUE" : "isprint(c) = FALSE");
     printf( ispunct(c) ?
             "ispunct(c) = TRUE" : "ispunct(c) = FALSE");

   }
}
```

Notes:

- Most libraries implement these functions as macros which access an array of character classes. See CTYPE.H and related documentation. The array method has a fixed overhead and portability going for it at the expense of code size (in some cases).

- All functions above are honest functions and not macros. There are no side effects, even for the ctype functions.

- If you use only isspace(), then including CTYPE.H will cost you dearly in code size. For a few istype functions it is more economical to use the standard functions in CLIB.REL.

The itoa Function

Purpose:

Converts an integer into a null-terminated string of decimal digits, with leading minus sign as appropriate.

Function Header:

```
char *itoa(x,s)
int x;                  Integer to be converted.
char *s;                Buffer for string of decimal digits.
```

Returns:

```
s                   Base address of the conversion string.
```

Example: Convert a data item to a string of digits and print.

```
#include <stdio.h>
main()
{
char s[20];
#define NUMBER 10133
     itoa(NUMBER,s);
     puts(s);
}
```

Notes:

- Unsigned integers greater than 32767 will be treated as long integer data by the compiler, unless a suitable cast is imposed on the number.
- The internal storage of an integer is two bytes (16 bits), stored in the usual 8080 reverse format. Integers typed at the terminal are in ascii decimal format, using the character set '0',...,'9'.
- No check is made to see if the storage area is large enough to accept the string . This is left to the programmer.
- Do not confuse this function with ltoa(), which converts a long signed integer to ASCII. The latter uses a 32-bit first argument.
- Numbers larger than 32767 are assumed to be long integers by the compiler. If you want to convert an unsigned integer to ASCII, then use the function sprintf() instead.

The keystat Function

Purpose:

Test the console buffer and the keyboard for a character available.

Function Header:

```
int keystat()
```

Returns:

```
    0         No character is available.
 nonzero      Character is available.
```

Example: Expunge the console buffer.

```
main(argc,argv) int argc; char **argv;
{
char s[129];
  puts("Press any key to start");
  getbyte();
  conflush();
  puts("\nEnter a line of text");
  gets(s);
  puts(s);
}

conflush()
{
  while(keystat()) getbyte();
}
```

Notes:

- Function keys can create havoc with character mode input. Use keystat() to tell whether or not a character is available.
- This keystat() is not the same as bdos(11,0). In addition, it checks the console type-ahead buffer at Cbuf.
- Scanf() can cause problems too. We suggest that you flush the console input buffer as in the example each time that scanf() fails to return the required number of arguments.

The labs Function

Purpose:

Compute the absolute value of a signed long integer quantity.

Function Header:

```
long labs(value)                    Long Integer Absolute value
long value;                         Long argument, 32 bits
```

Returns:

The absolute long integer value of the argument.

Example: Print the long integer absolute value of a number entered at the console.

```
#define pfLONG 1
#include <stdio.h>
main()
{
char s[129];
long a,labs();

     printf("Enter long integer a: ");
     gets(s); a = atol(s);
     printf("labs(%ld) = %ld\n",a,labs(a));
}
```

Notes:

- Not a macro. It often is a macro in other libraries.
- There are no side effects. Labs() is an honest function.
- The cast must appear explicitly in your program. For example, if you use labs(), then include a declaration of the form

  ```
  long labs();
  ```

- A common error is to write ln = abs(n) where ln,n are signed long integers. This has undefined results, but there is no compile-time error reporting.

The ldexp Function

Purpose:

Computes x*(radix**n) where n is an integer and the value of x is typically 0.5 <= x < 1.0.

Function Header:

```
float ldexp(x,n)
float x;            Float value in range 0.5 to 1.0
int n;              Integer exponent. Range -38 to +38 expected.
```

Returns:

```
x*(2**n)        Mantissa times power-two exponent.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
int n;
float x;
extern int errno;
float atof();
float ldexp();
  errno = 0;
  printf("Enter a float x: "); gets(s); x = atof(s);
  printf("Enter an integer n: "); gets(s); n = atoi(s);
  printf("ldexp(%f,%d) = %f, errno = %d\n",x,n,ldexp(x,n),errno);
}
```

Notes:

- The float exponent in this library is r = ((int)255 & (int)x.c[3]) - (int)128.

- A portable version of ldexp() that can be used to check an implementation appears below. The version in this library plugs the exponent in the float with a new value.

```
float ldexp(x,n) float x; int n;
{
int i; float f;
  i = n; f = x;
  if(i < 0) while(i++ < 0) f = f/2.0;
  else      while(i-- > 0) f += f;
  return f;
}
```

The natural log Function

Purpose:

Compute the logarithm base e of real float value x.

Function Header:

```
#include <math.h>
float ln(x)         Log base e. Defined in MATH.H as log().
float log(x)        Log base e, or Naperian Logarithm.
float x;            Positive float argument.
```

Returns:

```
number          x in range
INF             x too large positive
-INF            x = 0 or x < 0
                errno = EDOM returned to flag error
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float log();
     errno = 0;
     printf("Enter positive float x: "); gets(s); x = atof(s);
     printf("log(%f) = %f, errno = %d\n",x,log(x),errno);
}
```

Notes:

- Method used is ln(x) = log10(x)/log10(e) = log10(x)*ln(10).
- Not very well done. Errors must filter back through the log10() function.

The log10 Function

Purpose:

Compute the base 10 logarithm of real float value x.

Function Header:

```
float log10(x)      Log base 10, same as log(x) on most
                    calculators.
float x;            Positive float argument.
```

Returns:

```
number          x in range
INF             x too large positive
-INF            x = 0 or x < 0
                errno = EDOM returned to flag error
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float log10();
     errno = 0;
     printf("Enter positive float x: "); gets(s); x = atof(s);
     printf("log10(%f) = %f, errno = %d\n",x,log10(x),errno);
}
```

Notes:

o Method used is Rational approximation. See C&H 6.3.28. Constants from table 2325.

The logged Function

Purpose:

Reports the currently logged disk.

Function Header:

```
int logged();
```

Returns:

n	n = 0 to 15 for drives A: to P:

Example: Find out which disk drive is the default disk.

```
#include <stdio.h>
main()
{
int c;
  c = 'A'+logged();
  printf("Default drive is %c:\n",c);
}
```

Notes:

- Logged() is not a standard function. It is useful for writing interactive CP/M software.
- To simulate logged() on other systems, use the bdos() function or the ccBDOS() function.

```
#include <unix.h>
#include <stdio.h>
logged()
{
   return bdos(25,0);
}
```

```
#include <stdio.h>
logged()
{
   return ccBDOS(25);
}
```

The lsearch Function

Purpose:

Linear search and update of an arbitrary table of info. Requires a special compare function to match the internal table structure.

Function Header:

```
char *lsearch(key,base,n,w,cmp)
char *key;                  Key for table search.
char *base;                 Base address of sorted table.
int *n;                     Pointer to table size.
int w;                      Width of each table element.
int (*cmp)();               Compare function cmp(key,tbl)

int cmp(key,tbl);           User-supplied compare routine.
char *key;                  key = address of the key
char *tbl;                  tbl = address of table entry;
```

Returns:

```
(char *)      Location of table match
(char *)0     Not found
```

Example: Search and update a list of strings.

```
#include <stdio.h>
    static char *tbl[] = {
      "ABC","AB","EFG","HJK","M","abc","abd","bcf",
      "","","","
```

Notes:

- The table width is 2 = sizeof(char *). The strings are not all of the same width. The table tbl[] is a table of pointers, which causes the unusual addressing modes seen in the example. Extra null pointers were used for table expansion. The code must test availability of expansion space.

- This is a UNIX function. See also bsearch(), binary(), sbinary(), ssort2(), ssort3(), numsort(), dsort(), dsort16(), shells(), qsort(), quick(), heap().

- The example shows how to make permanent entries into an updated table. Sometimes this is not necessary, especially for numerical tables.

- The addressing modes required for string handling are unnatural. See the example above to get it right.

- The complete source follows. The most common application error involves the integer pointer n.

```
char *lsearch(key,base,n,w,cmp)
char *key;
char *base;
int  *n;
int  w;
int  (*cmp)();
{
static
int m,j;
#define TARGET base
#define SOURCE &key
#define LENGTH W
     m = *n;
     while(m-- > 0) {
       if((j = (*cmp)(key,base)) == 0)
         return base;
       base += w;
     }
     *n = (*n)+1;
     moveMEM(TARGET,SOURCE,LENGTH);
     return TARGET;
}
```

The loginv Function

Purpose:

Fetches the logged-disk vector, a 16-bit flag vector.

Function Header:

```
unsigned loginv()
```

Returns:

n	A 16-bit word, where bit i is set when drive i is logged. Drives 0 to 15 correspond to A: to P:.

Example: Find out which disk drives are currently logged in.

```
#include <stdio.h>
main()
{
int i;
unsigned n;
  n = loginv();
  for(i=0;i<16;++i) {
    if(n&1) printf("Drive %c: is logged in\n",i+'A');
    n = (n >> 1);
  }
}
```

Notes:

- Loginv() is not a standard function. It is important for writing interactive CP/M software.
- To simulate loginv() on other systems, use the ccBDOS() function:

```
#include <stdio.h>
unsigned loginv()
{
unsigned ccBDOS();
   return ccBDOS(24);
}
```

The lowmem Function

Purpose:

Computes the lowest memory address that is assignable by sbrk(). The answer returned is sbrk(0).

Function Header:

```
char *lowmem()
```

Returns:

The break address for the program, last assigned by sbrk(), or fixed at compile time by the program text and data, plus the run-time file buffers.

Example: Check a file open to see if it called sbrk().

```
#define nfiles 0
#include <stdio.h>
main()
{
char *lowmem();
char *p;
FILE *fp,*fopen();
     p = lowmem();
     if(fp = fopen("TMP","w")) {
       if(p == lowmem()) printf("No change in sbrk()\n");
       else printf("sbrk() was called during fopen()\n");
       fclose(fp);
       unlink("TMP");
     }
}
```

Notes:

- This function is left around as an artifact. To produce portable code, use sbrk(0) or core(0) instead.

The ltoa Function

Purpose:

Converts a 32-bit signed long integer into a null-terminated string of decimal digits, with leading minus sign as appropriate.

Function Header:

```
char *ltoa(x,s)
long x;                     Long integer to be converted.
char *s;                    Buffer for string of decimal digits.
```

Returns:

```
s                   Base address of the conversion string.
```

Example: Convert a data item to a string of digits and print.

```
#include <stdio.h>
main()
{
char *ltoa();
char s[20];
#define NUMBER (long)1010133
#define UNUMB  (long)100
     puts(ltoa(NUMBER,s));
     puts(ltoa(UNUMB,s));
}
```

Notes:

- Unsigned integers greater than 32767 will be treated as long integer data by the compiler, unless a suitable cast is imposed on the number.
- Signed integers from -32767 to 32767 will require a cast of (long) in order to be used as an argument to ltoa(). See the example.
- Long integers are 32 bits (4 bytes). Storage of a long integer is such that you may address its lower order 16 bits as an integer, without error. However, this is highly non-portable (68000 CPU, for example).
- The cast of ltoa() is (char *), which must be declared prior to usage as in the example above.

The makeFCB Function

Purpose:

Makes a CP/M file control block of 36 bytes as per interface standards. Expands * and ? wildcards.

Function Header:

```
VOID makeFCB(fcb,filename)
char fcb[36];                   File control block buffer.
char *filename;                 Name of file, null-terminated.
```

Returns:

Nothing useful. Fills the file control block with nulls, expands any wildcards * or ? in the file name to all question marks.

Example: Make a file control block from a wildcard file name.

```
#include <stdio.h>
main()
{
char fcb[36];
char *decodF();
  makeFCB(fcb,"A:*.TXT");
  printf("string = %s\n","A:*.TXT");
  printf("fcb = %s\n",decodF(fcb));
}
```

Notes:

- Decodf(fcb) decodes a file control block into a printable ASCII file name in the usual CP/M format.
- Unspecified areas in the filename and extension are blank-filled. Areas beyond the file name are filled with nulls as per CP/M interface standards. A 36-byte file control block is assumed.
- Classic mistakes in using makeFCB() include reversal of the arguments and the disaster of using a file control block shorter than the required 36 bytes.

The malloc Function

Purpose:

Malloc() allocates contiguous memory located between the end of the program and the stack. Free() releases memory assigned by malloc().

Function Header:

```
char *malloc(m)
unsigned m;                 Unsigned integer, amount of memory
                            that is requested, in bytes.
```

Returns:

The base address of the area, on success, 0 if the request fails.

Example: Get 2048 bytes from malloc(), then free it.

```
#include <unix.h>
#include <stdio.h>
getmem()
{
char *p;

     p = malloc(2048);
     if(p == (char *)0) puts("Request failed");
     else
     free(p);
}
```

Structure used by malloc(): The following 4-byte header appears just before the base address returned by malloc(). It is used by both malloc() and free(), so be careful not to corrupt it.

```
struct block {
        struct block
            *nxtblk;
        unsigned
                siz;
};
```

Notes:

- Uses sbrk() to get raw system memory.
- You can't ask for more than 65531 bytes.
- Use free() to release a block.
- Code below derived from K&R(1978), pp 174-177. A few mods were made to compact the code under C/80.
- Two successive calls to malloc() will not in general result in a single block of continuous memory.
- Calls to malloc() cause a header to be written at the beginning of the block. Malloc() uses the header in an essential way - don't write over it!
- Sfree() doesn't know about malloc(). Look out!
- Brk() doesn't know about malloc(). Look out!
- Memory obtained from sbrk() comes in hunks of size HEAPSIZE:

```
#define SIZBLOCK        sizeof(struct block)
#define HEAPSIZE        256*SIZBLOCK
```

  This implies that malloc() can fail even though sbrk() can assign more memory.

The max Function

Purpose:

Computes the maximum value of two integer arguments.

Function Header:

```
int max(x,y)
int x,y;                    Integers to compare.
```

Returns:

The larger of x and y as signed integers, i.e., max(-1,-2) = -1 and max(1000,2000) = 2000.

Example: Print the maximum value of two numbers a,b entered at the console.

```
#include <stdio.h>
main()
{
char s[129];
int a,b;

     printf("Enter number a: ");
     gets(s); a = atoi(s);
     printf("Enter number b: ");
     gets(s); b = atoi(s);
     printf("max(%d,%d) = %d\n",a,b,max(a,b));
}
```

Notes:

- The max() function used here is an honest function that works only on integers. It fails on long integers and floats.
- For long integers, use max = (a > b ? a : b).
- For floats, use max = (a > b ? a : b).
- Most libraries assume the max() function is a macro defined in the STDIO.H header file or in MATH.H. Such functions definitely have side effects. Beware when you port the code. Engineer against side effects when you write it for the first time.

The memsize Function

Purpose:

Computes the amount of free memory in the heap that is assignable by the next call to sbrk().

Function Header:

```
unsigned memsize()
```

Returns:

Unsigned integer in the range 0 to 65635, which represents the number of bytes free in the heap between the program end (as set by brk() or sbrk()) and the top of the user stack: memsize() = highmem()-lowmem().

Example: Find the size of the biggest possible text buffer and use it to load a file into memory with raw read.

```
#define nfiles 1
#define fd     fp              /* fd = fileno(fp), see fileno() */
#include <stdio.h>
main()
{
unsigned amt;
char *core();
FILE *fp,*fopen();
     amt = memsize();
     p = core(amt);
     printf("Internal buffer size %u\n",amt);
     if(fp = fopen("TMP","w")) {
       amt = read(fd,p,amt);
       printf("Read %u bytes into the internal buffer\n",amt);
     }
}
```

Notes:

- This function is left around as an artifact. It helps to build a portable interface that allows code to be transported easily. Note that memsize() can be written for another system to return a fixed amount like 110000.

The midstr Function

Purpose:

Finds the position (1...32767) at which str2 matches str1.

Function Header:

```
int midstr(str1,str2)
char *str1;              Source string.
char *str2;              String to match.
```

Returns:

```
k>0       If str2 matches str1 starting at offset k-1.
0         No match.
```

Example: Test program for properties of midstr(). Reports timing differences between midstr() and instr().

```
#include <stdio.h>
main()
{
int i,j;
char str1[100],str2[100];
 for(;;) {
 fputs("Enter str1: ",stderr);
 gets(str1);
 fputs("Enter str2: ",stderr);
 gets(str2);
 timer(0);
 for(j=0;j<200;++j)
  i = midstr(str1,str2);
 timer(1);
 printf("midstr = %d\n",i);
 timer(0);
 for(j=0;j<200;++j)
  i = instr(1,str1,str2);
 timer(1);
 printf("instr = %d\n",i);
 }
}
```

Notes:

- The first use for midstr() is to replicate the two-argument INSTR(A$,B$) found in Microsoft's MBASIC programming language.

- This function is two to six times faster than instr(). Use it where speed counts, like in editor searches.

- A portable version of midstr() appears below, in case you need to port the code to a different target. The portability causes the speed to be lost, but it is useful for bringing up a program for the first time. Fast implementations should not call strlen() or strncmp(). Assembler is mandatory even on a 10mhz 68000 CPU.

```
int midstr(s,t)
char *s,*t;
{
int i,j;
        i = strlen(t);
        j = 0;
        while(*s) {
             ++j;
             if(*s == *t && strncmp(s,t,i) == 0) return j;
             ++s;
        }
        return 0;
}
```

The min Function

Purpose:

Computes the minimum value of two integer arguments.

Function Header:

```
int min(x,y)
int x,y;                    Integers to compare.
```

Returns:

The smaller of x and y as signed integers, i.e., min(-1,-2) = -2 and min(1000,2000) = 1000.

Example: Print the minimum value of two numbers a,b entered at the console.

```
#include <stdio.h>
main()
{
char s[129];
int a,b;

     printf("Enter number a: ");
     gets(s); a = atoi(s);
     printf("Enter number b: ");
     gets(s); b = atoi(s);
     printf("min(%d,%d) = %d\n",a,b,min(a,b));
}
```

Notes:

- The min() function used here is an honest function that works only on integers. It fails on long integers and floats.
- For long integers, use min = (a < b ? a : b).
- For floats, use min = (a < b ? a : b).
- Most libraries assume the min() function is a macro. Look in the STDIO.H header file or in MATH.H. Such functions definitely have side effects.

The mktemp Function

Purpose:

Create new contents for the last 6 characters of a null-delimited string. The string can be used as a filename (up to 8 chars, no extension).

Function Header:

```
char *mktemp(tmp)
char *tmp;          Address of the template string to be altered.
                    The argument should be a NULL-terminated string
                    which ends in the six special characters
                    XXXXXX.
```

Returns:

Its argument, the string base address. Generally, the return value is not useful.

Example: Open a temporary file, write one byte, close it and delete the file from the directory.

```
VOID testtmp()
{
FILE *fopen();
FILE *fp;
char *p,*mktemp();
char name[20];

    strcpy(name,"@@XXXXXX");
    mktemp(name);
    strcat(name,".TMP");
    if(fp = fopen(name,"r")) {
            puts("TMP file already in use");
            exit();
    }
    if(fp = fopen(name,"w")) {
        putc('X',fp);
        fclose(fp);
    }
}
```

Notes:

- It is up to the caller to supply a string as requested.
- The filename is derived by writing ascii digits over the XXXXXX portion of the string. Two leading characters are optional. An extension can be added using strcat().

The modf Function

Purpose:

Splits float x = f + n, where n is an integer and float f satisfies fabs(f) < 1.0.

Function Header:

```
float modf(x,nptr)
float x;                Float to split.
int *nptr;              Where to put the exponent.
```

Returns:

```
f           The fraction, a float remainder.
n           via *nptr = n;
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,f;
int n;
float atof();
float modf();
     printf("Enter float x: "); gets(s); x = atof(s);
     f = modf(x,&n);
     printf("modf(%f,nptr) = %f, *nptr = %d\n",x,f,n);
}
```

Notes:

- No error codes returned. No need to code for errno.
- No overflow check.
- No auto conversion to FLOAT.

The moveMEM Function

Purpose:

Copies one region of memory to another without ripple.

Function Header:

```
char *moveMEM(dest,source,n)
char *dest;                 Destination address.
char *source;               Source address.
unsigned n;                 Number of bytes to fill.
```

Returns:

```
dest+n                      Next location after move.
```

Example: Copy a buffer to another location.

```
#include <stdio.h>
main()
{
char s[100];
char t[100];
     fillchr(s,'A',100);
     moveMEM(t,s,100);
}
```

Notes:

- Usage of this function differs among libraries. Often called movmem() with arguments in a different order.
- The order of moveMEM arguments is the same as strncpy() and strncat(), which is in turn the same order as fillchr(). This consistency evidently escaped the authors of other libraries.
- No ripple means that the buffers being moved may be overlapping. There is no danger in using this function, whereas strncpy() may have problems.
- In this library, auto-switching to a Z80 block move is made on capable CPU's. If the Z80 CPU test fails, then a standard 8080 block move is done.

The movmem Function

Purpose:

Non-ripple memory move. BDS-C, CI-86, WIZARD and LATTICE standard.

Function Header:

```
VOID movmem(source,dest,n)
char *source;                   Source address.
char *dest;                     Destination address.
int n;                          Number of bytes to move.
```

Returns:

Nothing.

Example: Copy a file control block and print it.

```
#include <unix.h>
#include <stdio.h>

main()
{
char.fcb[36];
extern char *IOfcb[];
int fd,i;
FILE *fp,*fopen();

     fp = fopen("TMP","w");
     if(fp) {
          fd = fileno(fp);
          movmem(IOfcb[fd],fcb,36);
          fclose(fp); unlink("TMP");
          for(i=0;i<36;++i) printf("fcb[%d] = %u\n",i,fcb[i]);
     }
}
```

Notes:

- This function calls the main library function called moveMEM. The argument order of moveMEM() is the same as strncpy(). The argument order of movmem() has the destination and source reversed from strncpy() and strncat().

- A portable version of movmem exists, but it is slow:

```
movmem(s,d,n)
char *s,*d; int n;
{
int m;
   if(d<s) {
        while(n--) *d++ = *s++;
   }
   else {
        d += n; s += n;
        while(n--) *--d = *--s;
   }
}
```

- The above source is the same as moveMEM(d,s,n). Both functions are generally optimized. On an 8088/8086 machine, the string primitives of the CPU are used. On a Z80 machine, the CPU block memory moves are used.

The numsort Function

Purpose:

Distribution sort for numbers in High-Speed Assembler. Implements Donald Knuth's MathSort algorithm for 16-bit numbers.

Function Header:

```
int numsort(n,table,scrap)
int n;                          Number of integers to sort.
int table[];                    Array of integer data. Minimum
                                size of the table is 2n bytes.
int scrap[];                    Array, scratch space, 2n+512
                                bytes minimum.
```

Returns:

0 Success.

Example: Sort a list of integers.

```
#include <stdio.h>
main()
{
int i;
static
int data[] = {
     9324,2352,3243,6556,5554,
     4445,4544,5421,4221,4343
};
auto int scrap[10+256];
   if(numsort(10,data,scrap) == 0)
     for(i=0;i<10;++i) printf("%u\n",data[i]);
}
```

Sample output:

```
2352
3243
4221
4343
4445
4544
5421
5554
6556
9324
```

Notes:

- The order of the arguments is essential. Actually, the integer n can be unsigned, as far as the assembler code is concerned. But the address space of an 8080 machine will not allow more than 32767 integers.
- Sorts 12000 numbers in 3 seconds at 4mhz on a Z80 machine using Digital Research CP/M 2.2.
- This routine can only be used on 16-bit data.
- The source code is written in 8080 assembler, but full C source appears in the source code comments. See the source archives.
- The use of an auto array can cause trouble. The total byte count for all auto arrays in a function cannot exceed 32767. If in doubt, then use sbrk() (or malloc()) followed by sfree() (or free()).
- The scrap array is used by numsort() but it does not return anything useful. The sorted data is returned in table[].

The open, opena Functions

Purpose:

Ascii File open() by file descriptor.

Function Header:

```
#include <unix.h>
int  open(name,access)
char *name;              An ascii NULL-terminated filename string
int   access;            Access, an integer 0,1,2. ASCII text mode
                         only, as follows:

                         0       Read-only
                         1       Write-only
                         2       Update (read & write)
```

Returns:

```
-1      Open failed - not the same as fopen!
fd      Open worked, fd=file descriptor
```

WARNING: In UNIX.H the following definition is made:

```
#define open opena
```

Note that opena(), openb() are real functions and not macro definitions.

Example: Open a disk file by descriptor and write 128 bytes from a buffer.

```
#include <unix.h>
#include <stdio.h>
pump(buffer,file)
char *buffer,*file;
{
int fd;
     fd = open(file,1);        /* open for output */
     if(fd == -1) {
          puts("Bad open"); return;
     }
     write(fd,buffer,128);
     close(fd);
}
```

Notes:

- Use of this function avoids seekend() and its corresponding overhead. Append access is not an option.
- The CP/M devices "CON:", "RDR:", "LST:", "PUN:" can be opened by using this function. They are unbuffered files. Console input ha a type-ahead buffer, which makes it a hybrid unbuffered file.
- Binary mode is supported through openb().
- Beware of the lack of portability of binary mode. When implemented, many libraries try to do so by extended coding of the access. The latter is the worst of choices since it transports invisible program bugs.

The openb Function

Purpose:

Binary File open() by file descriptor.

Function Header:

```
int  openb(name,access)
char *name;             An ascii NULL-terminated filename string
int   access;           Access, an integer 0,1,2. BINARY text mode
                        only, as follows:

                        0       Read-only
                        1       Write-only
                        2       Update (read & write)
```

Returns:

```
-1      Open failed - not the same as fopen!
fd      Open worked, fd=file descriptor
```

WARNING: In UNIX.H the following definition is made:

```
#define open opena
```

Note that opena(), openb() are real functions and not macro definitions.

Example: Open a disk file by descriptor and write 128 bytes from a buffer. The write will not translate CR/LF pairs.

```
#include <unix.h>
#include <stdio.h>
pump(buffer,file)
char *buffer,*file;
{
int fd;
     fd = open(file,1);        /* open for output */
     if(fd == -1) {
          puts("Bad open"); return;
     }
     write(fd,buffer,128);
     close(fd);
}
```

Notes:

- Use of this function avoids seekend() and its corresponding overhead. Append access is not an option.
- The CP/M devices "CON:", "RDR:", "LST:", "PUN:" can be opened by using this function. They are unbuffered files. Console input has a type-ahead buffer, which makes it a hybrid unbuffered file.
- Binary support for the devices does not really exist. The console can be put into binary mode by using the function CHmode(). This is a CP/M kludge and not a portable feature.
- Beware of the lack of portability of binary access. When implemented, many libraries try to do so by extended coding of the access. The latter is the worst of choices since it transports invisible program bugs.

The outport Function

Purpose:

Sends a value out a given CPU port.

Function Header:

```
int outport(port,value)
int port;                    Port number.
int value;                   Value to transmit.
```

Returns:

Nothing useful. Assumes the port is ready to receive a character. See the application notes and the following example.

Example: Send characters out the modem port.

```
#define MODEM      0330  /* Z90A computer, Z80 CPU, 255 ports */
#define OFFSET     0005  /* 8250 UART, line status register */
#define ISEMPTY    0040  /* test transmitter register empty */
#include <stdio.h>
main()
{
     fprintf(stderr,"Enter characters, end with ctrl-Z\n");
     while((x = getchar()) != EOF) {
          /* get char, wait for status, ship the byte */
          while((inport(MODEM+OFFSET)&ISEMPTY) == 0) ;
          outport(MODEM,x);
     }
}
```

Notes:

- On most machines the status of the port must be interrogated in order to write to the data port. This is true regardless of the port buffering.
- Disk I/O ports are often interrupt driven, so writing to the port will require set-up of the interrupt vector for return.
- Clock and timer chips like the 8253 Intel make for good examples of port usage in C programs. See also Analog-to-Digital boards and parallel port interfaces. These appear on music boards and game interfaces. The signals are joysticks, mice, light pens, plotters, graphics tablets and the like.
- Port operations are inherently machine-dependent. Most systems people don't care, however, because the problem being solved by port operations is usually special and one-time.

The Peekb, Peekw and Peekl Functions

Purpose:

Fetch a value from main memory, either 8, 16 or 32 bits.

Function Header:

```
char peekb(addr);
char *addr;                         Byte address.

unsigned peekw(addr);
char *addr;                         Int address.

long peekl(addr);
char *addr;                         Long integer address.
```

Returns:

```
(char)peekb()       Byte at address
(int)peekw()        Word at address
(long)peekl()       Long integer at address
```

Example: Access a long integer in three ways.

```
#define pfLONG 1
#include <stdio.h>
main()
{
char *addr;
long x;
char peekb(); unsigned peekw(); long peekl();

  x = 238823492;
  printf("x = 238823492\n");
  printf("byte=%c\n",peekb(&x));
  printf("word=%u\n",peekw(&x));
  printf("long=%ld\n",peekl(&x));
}
```

Notes:

- o These functions are used when it is clumsy to use pointers.
- o Pointers sometimes hide ideas. These functions can help to document otherwise obscure code.
- o Beware of using peekl() without its proper declaration.

The perror Function

Purpose:

Writes a short message on the system console, describing the last system error, as detected by the variable errno. The message is derived from the string table sys_errlist[].

Function Header:

```
#include <errno.h>
char *sys_errlist[];    Error strings. See below.
int sys_nerr;           Number of error strings.
int  perror(s)
char *s;                Prefix string to print prior to message.
```

Returns:

The value of errno before the call.

Example: Print the current error from variable errno.

```
#include <unix.h>
#include <math.h>
#define pfFLOAT 1
#include <stdio.h>
main()
{
float sin();
extern int errno;
     errno = 0;
     printf("sin(%f) = %f\n",5600.01,sin(5600.01));
     if(errno) perror("sin function error: ");
}
```

Notes:

- Not much of the library currently uses the errno variable. Mostly for the transcendental functions.
- Fixes to the library should go through this routine during debugging to insure that the proper messages get printed.
- The currently supported error message appear below:

```
char *sys_errlist[] = {
"Errno 0 detected",
"Permission denied",
"File not found",
"",     /* No such process */
"",     /* System call interrupted by signal */
"I/O error",
"No such I/O device",
"",     /* Too many arguments to exec */
"",     /* Wrong format for executable file */
"Bad file descriptor",
"",     /* No children */
"",     /* Cannot fork */
"Too little memory available",
"",     /* File access conflict with user rights */
"",     /* Bad memory address supplied */
"",     /* Wrong sort of device */
"",     /* Device already in use */
"The file already exists",
"",     /* You cannot link across devices */
"",     /* Silly access to this device */
"",     /* Directory name expected */
"",     /* Directory name not expected */
"",     /* Invalid argument */
"",     /* System out of file table space */
"",     /* Request for too many file descriptors */
"",     /* Not a teletype device */
"",     /* File currently in use */
"",     /* File became too large for system */
"Disk system out of space",
"",     /* Seek attempted on pipe */
"",     /* Read-only file system */
"",     /* Too many links to a file */
"",     /* Write attempt on broken pipe */
"Out of function domain",
"Result out of range"
};

int sys_nerr = sizeof(sys_errlist)/sizeof(sys_errlist[0]);
```

The Pokeb, Pokew and Pokel Functions

Purpose:

Store a value to main memory, either 8, 16 or 32 bits.

Function Header:

```
pokeb(addr,x);
char *addr;                 Address to store the byte
char x;                     8-bit data to be stored

pokew(addr,x);
char *addr;                 Address to store the word
unsigned x;                 16-bit data to be stored

pokel(addr,x);
char *addr;                 Address to store the long integer
long x;                     32-bit data to be stored
```

Returns:

Nothing useful

Example: Union manipulation using the poke functions.

```
#define pfLONG 1
#include <stdio.h>
main()
{
union { char c; int i; long x; } xx;

  printf("Answers should be: 65, 65000, 6500000\n");
  pokeb(&xx.c,(char)65);
  printf("character xx.c=%u\n",xx.c);
  pokew(&xx.i,(unsigned)65000);
  printf("integer xx.i=%u\n",xx.i);
  pokel(&xx.x,(long)6500000);
  printf("long xx.x=%ld\n",xx.x);
}
```

Notes:

- These functions are used when it is clumsy to use pointers.
- Pointers sometimes hide ideas. These functions can help to document otherwise obscure code.
- A typical application in systems is pokew(peekw(1)-16). While it can be done with pointers, the above is easier to debug.

The pow Function

Purpose:

Compute the power of real float values x,y i.e., computes x raised to power y.

Function Header:

```
float pow(x,y)      Standard function x^y.
float x,y;          Base = x, exponent = y.
```

Returns:

```
number          x and y in range
1.0             x nonzero and y = 0
INF             arguments give answer too large
                errno = ERANGE returned to flag error
-INF            x < 0 and y not an integer
-INF            x = 0 and y <= 0
                errno = EDOM returned to flag error
-pow(-x,y)      x < 0 and y = odd integer
 pow(-x,y)      x < 0 and y = even integer
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x,y;
extern int errno;
float atof();
float pow();
     errno = 0;
     printf("Enter float base x: "); gets(s); x = atof(s);
     printf("Enter float exponent y: "); gets(s); y = atof(s);
     printf("pow(%f,%f) = %f, errno = %d\n",x,y,pow(x,y),errno);
}
```

Notes:

- Table lookup. See C&H 6.2.34. Tables from C&H 1403.
- This function tends to be a bit slow.
- pow() uses the power of 10 function pow10(), which does limited error checking.

The pow10 Function

Purpose:

Compute power base 10 of real float value y i.e., computes 10 raised to power y.

Function Header:

```
float pow10(y)      Standard function 10^y.
float y;            Base = 10, exponent = y.
```

Returns:

```
number              Assume y in range.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float y;
extern int errno;
float atof();
float pow10();
     errno = 0;
     printf("Enter float exponent y: "); gets(s); y = atof(s);
     printf("pow10(%f) = %f, errno = %d\n",y,pow10(y),errno);
}
```

Notes:

- This function does almost no error checking because the true front end is the pow() function. This part of the code is slow already, and any additional error checking seems not to be needed in a float-only library.

The printf Function

Purpose:

Performs output formatting for the standard output stream stdout. Uses a control string followed by appropriate arguments.

Function Header:

```
int printf(control,arg1,arg2,...)
char *control;              Control string, see below.
arg1,arg2,...               Appropriate arguments, see below.
```

The control string contains:

White space	Tabs, spaces, return, formfeed, vertical tab, newline, backspace
Escapes	Octal values prefixed by \, e.g., \007
Characters	Ordinary printing characters other than %
%%	Prints as %
%	Conversion flag

Following a conversion flag %, there will be an optional FLAG CHARACTER:

1. minus sign, signifies left-justification rather than right.
2. zero, means pad with zero rather than spaces.

Harbison & Steele recommend other modifiers, which are not supported:

3. plus sign, means always output + or - with numbers.
4. space, means always produce - or space fill.
5. #, means use a variant of the main conversion operation.

Following the flag character:

6. Field width, a sequence of decimal digits. Conversions are padded to this width, unless the field width is exceeded, in which case the whole field is printed and padding is ignored.

Following the field width:

7. An optional period and string of decimal digits, whose value is used to control the number of floating-point digits to the right of the decimal point. This called the Precision Field.

Following the precision field:

8. An optional LONG specifier, expressed as lowercase l, which is used where such a prefix makes sense. This is called the Long Prefix Field.

Following the long prefix field:

9. A conversion operation, expressed as a single character taken from the lowercase letters:

c	character
s	string
d	signed decimal base 10
u	unsigned decimal base 10
o	octal base 8
x	hexadecimal base 16
b	binary base 2
e	scientific format float, exponents
f	float format without exponents
g	smaller of e and f formats, strip zeros

The conversions E, G, X are not supported. The possible long conversions are:

ld	long signed decimal base 10
lo	long octal base 8
lx	long hexadecimal base 16
lb	long binary base 2
lu	long unsigned decimal base 10

Hexadecimal conversions use capital letters only. Binary conversion is provided as a debugging convenience. Do not expect it to be a portable feature of printf. All scientific exponents use lower case e rather than E.

Returns:

EOF	Error occurred for output
?	No standard exists for return otherwise. Probably should be the number of characters printed.

Under C/80, extra parentheses are required to capture the return value of printf, e.g., (printf(...)). But CP/M cannot tolerate I/O on a full disk, so the library gracefully warm boots, long before you could capture an EOF indication.

Example: Control of field width.

```
#define mathlib 1
#include <stdio.h>
#undef printf
#define printf prnt_1(),prnt_2          /* select portable printf */
main()
{
static char *s = "Hello World";
static char c = 'A';
  printf("%10s :%10s:\n",":%10s:",s);
  printf("%10s :%-10s:\n",":%-10s:",s);
  printf("%10s :%20s:\n",":%20s:",s);
  printf("%10s :%-20s:\n",":%-20s:",s);
  printf("%10s :%20.10s:\n",":%20.10s:",s);
  printf("%10s :%-20.10s:\n",":%-20.10s:",s);
  printf("%10s :%.10s:\n",":%.10s:",s);
  printf("%10s :%c:\n",":%c:",c);
  printf("%10s :%10c:\n",":%10c:",c);
  printf("%10s :%-10c:\n",":%-10c:",c);
  printf("%10s :%010c:\n",":%010c:",c);
  printf("%10s :%-010c:\n",":%-010c:",c);
  printf("%10s :%20s:\n",":%20s:","");
}
```

```
    :%10s: :Hello World:
   :%-10s: :Hello World:
    :%20s: :          Hello World:
   :%-20s: :Hello World          :
 :%20.10s: :          Hello World:
:%-20.10s: :Hello Worl           :
   :%.10s: :Hello Worl:
      :%c: :A:
    :%10c: :          A:
   :%-10c: :A          :
   :%010c: :000000000A:
  :%-010c: :A000000000:
    :%20s: :                    :
```

In this library, precision is the string length of the conversion regardless of the selected precision. As stated in K&R, precision takes effect only for float numbers and strings.

Example: Control of decimal padding character (0 or space).

```
#define mathlib 1
#include <stdio.h>
#undef printf
#define printf prnt_1(),prnt_2          /* select portable printf */
main()
{
static int x = 1001;
    printf("%10s :%10d:\n",":%10d:",x);
    printf("%10s :%-10d:\n",":%-10d:",x);
    printf("%10s :%010d:\n",":%010d:",x);
    printf("%10s :%10x:\n",":%10x:",x);
    printf("%10s :%-10x:\n",":%-10x:",x);
    printf("%10s :%010x:\n",":%010x:",x);
}
```

```
Output of the program:
:%10d: :      1001:
:%-10d: :1001      :
:%010d: :0000001001:
:%10x: :       3E9:
:%-10x: :3E9       :
:%010x: :00000003E9:
```

Example: Control of floating point precision.

```
#define pfFLOAT 1
#include <stdio.h>
main()
{
static float f = 45.3483;
    printf("%10s :%8.6f:\n",":%8.6f:",f);
    printf("%10s :%-8.6f:\n",":%-8.6f:",f);
    printf("%10s :%08.6f:\n",":%08.6f:",f);
    printf("%10s :%20.10f:\n",":%20.10f:",f);
    printf("%10s :%-20.10f:\n",":%-20.10f:",f);
    printf("%10s :%020.10f:\n",":%020.10f:",f);
    printf("%10s :%8.6e:\n",":%8.6e:",f);
    printf("%10s :%-8.6e:\n",":%-8.6e:",f);
    printf("%10s :%08.6e:\n",":%08.6e:",f);
    printf("%10s :%20.10e:\n",":%20.10e:",f);
    printf("%10s :%-20.10e:\n",":%-20.10e:",f);
    printf("%10s :%020.10e:\n",":%020.10e:",f);
    printf("%10s :%8.6g:\n",":%8.6g:",f);
    printf("%10s :%-8.6g:\n",":%-8.6g:",f);
    printf("%10s :%08.6g:\n",":%08.6g:",f);
    printf("%10s :%20.10g:\n",":%20.10g:",f);
    printf("%10s :%-20.10g:\n",":%-20.10g:",f);
    printf("%10s :%020.10g:\n",":%020.10g:",f);
}
```

Output of the program:

```
   :%8.6f: :45.348300:
  :%-8.6f: :45.348300:
  :%08.6f: :45.348300:
 :%20.10f: :       45.3483000000:
:%-20.10f: :45.3483000000       :
:%020.10f: :000000045.3483000000:
   :%8.6e: :4.534830e+01:
  :%-8.6e: :4.534830e+01:
  :%08.6e: :4.534830e+01:
 :%20.10e: :    4.5348300000e+01:
:%-20.10e: :4.5348300000e+01    :
:%020.10e: :00004.5348300000e+01:
   :%8.6g: : 45.3483:
  :%-8.6g: :45.3483 :
  :%08.6g: :045.3483:
 :%20.10g: :             45.3483:
:%-20.10g: :45.3483             :
:%020.10g: :00000000000045.3483:
```

Notes:

- Printf is not recursive. This means that you cannot have the result of a printf, fprintf or sprintf function call in the argument list for printf.
- The recursion loss is in the #define kludge for multiple arguments. The stack location is stored in a static variable and not on the stack, so repeated calls over-write the stack location.
- The small printf in the main library does have long or float support and does not pretend to be K&R standard, although it comes very close. Its advantage is speed, and to some extent, recursion.
- Binary conversion %b is not supported by very many libraries, however it is an invaluable debugging tool, especially in view of the lack of bit fields in C/80. Try to restrain yourself and use it only in debug code.
- The fat for printf comes largely from support code for long and float data types. See the examples above for switching code to turn on just what you need in your application.

The putc Function

Purpose:

Writes a character to an open stream.

Function Header:

```
int putc(x,fp)
int x;                  Character to output
FILE *fp;               Open stream pointer
```

Returns:

-1	If end of media was reached. We define EOF to be -1. End of file on output is either an error or end of media (out of disk space).
c	Character to be written (x), on success.

Example: The following writes characters to a file entered on the command line until ctrl-Z is entered. The explicit use of fclose() is required to enter the file information into the disk directory and flush any orphan record to disk.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
int x;
FILE *fp,*fopen();
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"w")) == (FILE *)0) {
          puts("Open failure"); exit(0);
     }
     while((x = getchar()) != EOF) {
          putc(x,fp);
     }
     fclose(fp);
}
```

Notes:

- This function is not a macro in the present library. Expect it to be a macro in most C libraries. It has no side effects.

- Under CP/M, the BIOS will complain on a disk write error, which is the only possibility besides end of file. In this case, the system will kill the running program unless you insist upon continuing with the error.

- A return of -1 under C/80 means that end of media was encountered. Otherwise, expect the character to be returned. Output to devices PUN: and LST: are expected to be subject to CR/LF translation unless done in binary mode. See putcbinary() and writeb().

- Beware of mixing file descriptors and stream pointers. While C/80 will buy it, other systems won't. The connection is fd = fileno(fp), where fd is an integer and fp is a stream pointer.

- C/80 will not allow an output to a full disk. Generally, putc() fails because of no disk space, and you will be bounced out to the system.

The putcbinary Function

Purpose:

Writes a character to an open stream in binary mode. Not a K&R function. Recognizes CON:, LST:, PUN: devices.

Function Header:

```
int putcbinary(x,fp)
int x;                  Character to output
FILE *fp;               Open stream pointer
```

Returns:

-1	If end of file was reached. We define EOF to be -1. For output, EOF means end of media.
c	Character to be written (x), on success.

Example: The following writes characters to stream stdout until the user types ctrl-Z. All characters are written in binary mode.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
int x;
     while((x = getchar()) != EOF) {
          putcbinary(x,stdout);
     }
}
```

Notes:

- A return of -1 under C/80 means that end of media was encountered. Otherwise, expect the character to be returned. Output to devices CON:, PUN: and LST: are expected to be subject to CR/LF translation.

- This function is a library internal, documented for your convenience. Do not expect it to be present on other systems.

- Since RDR: is a read-only device, putcbinary() will come back with an error for that stream. Use PUN:, also called AXO:.

- It is more portable to use putcbinary() than to constantly use the non-portable bios() features. Most target machines will support a function like putcbinary(), but the details of how to access binary output mode will vary across machines.

The putchar Function

Purpose:

Writes a character to the standard output stream stdout.

Function Header:

```
int putchar()
```

Returns:

c	Character output to stream stdout, on success.
-1	If an output error occurred, such as no disk space.

Example: The following writes characters to the console until ctrl-Z is encountered.

```
#include <stdio.h>
main()
{
int x;
     while((x = getchar()) != EOF) {
          putchar(x);
     }
}
```

Notes:

- Expect putchar to be a macro in most C libraries. It has no side effects. In the present library it is an honest function.
- Under CP/M, the BIOS will complain on a disk write error, which is the only possibility.
- A return of -1 under C/80 means that an error was encountered.
- putchar() writes to stream stdout, which may in fact be a file due to re-direction.
- If stream stdout is the console CON: or the punch PUN: (modem), then CR/LF translation is assumed. To disable this feature, use writeb() or putcbinary(). Under Bell Labs Unix systems, a newline character is used instead of the CP/M CR/LF pair, hence the need to convert will not exist on such systems.

The putl Function

Purpose:

Writes a 32-bit long integer to the stream, in Intel Reverse Format.

Function Header:

```
long putl(x,fp)
long x;                 32-bit integer to output to file
FILE *fp;               Stream pointer, open stream
```

Returns:

```
x       32-bit word, on success
-1      Error or end of file
```

Example: Write out long integers to a file.

```
VOID dumplongs(fp,fi,n)
FILE *fp;                       Open output stream
FILE *fi;                       Open input stream
int n;                          Number of elements
{
long x,putl(),getl();
  while(n) {
        if((x = getl(fi)) == EOF) {
            if(feof(fi) == EOF) break;
        }
        if(putl(x,fp) == EOF) {
            if(x != EOF) break;
        }
        --n;
  }
}
```

Notes:

- To detect end of media, check putl(x,fp) != x && x != -1.
- Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. Other libraries may use putc() to write putl(), thereby allowing translation of characters data corruption.
- Files opened for ASCII text mode usually cannot output a long integer using putc() or fputc() due to the danger of CR/LF translation. The latter is turned off during calls to putl() by using a special library function **putcbinary()**.

The puts Function

Purpose:

Writes characters to open stream stdout and appends a newline.

Function Header:

```
int puts(s)
char *s;                Base address of the string to be output.
```

Returns:

EOF	Error occurred.
c	Success, last character c which was output.

Example: A version of ECHO.C.

```
#include <stdio.h>
main()
{
char *gets();
char s[129];
  while(TRUE) {
    if(stdin == (FILE *)0)
    fprintf(stderr,"Enter a string or ctrl-Z to exit: ");
    if(gets(s) == (char *)0) break;
    puts(s);
  }
}
```

Notes:

- An error for puts() is usually a disk space error, therefore the library will dump the user to the system. CP/M will not tolerate I/O to a full disk.

- puts() differs from fputs() in that it appends a newline after the string is output. This newline is in addition to any newline that might be in the string.

The putw Function

Purpose:

Writes a 16-bit word to the stream, in Intel Reverse Format.

Function Header:

```
unsigned putw(n,fp)
unsigned n;                 16-bit unsigned integer to output
FILE *fp;                   Stream pointer for the output file
```

Returns:

```
n       16-bit word, on success.
-1      Error (write-protected) or end of media.
```

Example: Write words to a file and check for end of media.

```
pumpword(fp,x)
FILE *fp;
unsigned x;
{
     if(putw(x,fp) == EOF) {
        if(x != EOF) return -1;
     }
     return 0;
}
```

Notes:

- Beware of this function on other machines. It hides the byte sex problem. See for example CP/M-68K on the Motorola 68000 processor. It may be possible that another target machine uses putc() to write putw(), in which case character translation can occur during output.

- Files opened for ASCII text mode usually cannot output an integer using putc() or fputc() due to the danger of CR/LF translation. The latter is turned off during calls to putw() because of the explicit use of the special library function putcbinary().

The qsort Function

Purpose:

Quicksort with center pivot, stack control, and easy-to-change comparison method.

This version sorts fixed-length data items. It is ideal for integers, longs, floats and packed string data without delimiters.

Function Header:

```
int  qsort(base,n,s,CMP)
char *base;                Base address of the raw string data
int  n;                    Number of blocks to sort
int  s;                    Number of bytes in each block
int  (*CMP)();             Compare routine for two block pointers
                           p,q that returns an integer with the
                           same rules used by Unix strcmp(p,q):

                              = 0     Blocks p,q are equal
                              < 0     p < q
                              > 0     p > q
```

Beware of using ordinary strcmp() - it requires a NULL at the end of each string.

Returns:

0 Always

Example: Sort an array of integers.

```
#include <unix.h>
#include <stdio.h>
int mycmp(p,q) int *p,*q; { return (*p - *q);}
int q[10] = {12,1,3,-2,16,7,9,34,-90,10};
int p[10] = {12,1,3,-2,16,7,9,34,-90,10};
main()
{
int i;
        qsort(p,10,2,mycmp);
        for(i=0;i<10;++i) printf("%d. %d, %d\n",i,p[i],q[i]);
}
```

Output from the above sample program

```
0. -90, 12
1. -2, 1
2. 1, 3
3. 3, -2
4. 7, 16
5. 9, 7
6. 10, 9
7. 12, 34
8. 16, -90
9. 34, 10
```

Notes:

- Qsort() can sort raw integers, longs, floats or strings. However, the string sort is not efficient.
- Use quick() to sort string pointer arrays.
- Use cmpi(), cmpl(), cmpf(), cmps() to compare integers, longs, floats and strings, respectively.

References:

BYTE Oct-84, p 369
By William M. Raike
B*Y*T*E J*A*P*A*N

The Unix Book, p 200
by Banahan & Rutter
John Wiley & Sons, NY (1983)

The quicksort Function

Purpose:

Quicksort with center pivot, stack control, and easy-to-change comparison method.

Function Header:

```
quick(lo,hi,base,CMP)
int  lo;              First array subscript
int  hi;              Last array subscript
char *base[];         Base address of pointer array
int  (*CMP)();        Compare routine for two strings p,q
                      that returns an integer with the
                      same rules used by Unix strcmp(p,q):

                         = 0     strings p,q are equal
                         < 0     p < q
                         > 0     p > q
```

Returns:

0 Always

Example: Sort an array of strings.

```
char *p[10] = {"a","C","d","0","9","2","1","3","8","5"};
char *q[10] = {"a","C","d","0","9","2","1","3","8","5"};
int strcmp();
#include <unix.h>
#include <stdio.h>
main()
{
int i;
       quick(0,9,p,strcmp);
       for(i=0;i<10;++i) printf("%d. %s, %s\n",i,p[i],q[i]);
}
```

Results from the above program

```
0. 0, a            Numerals are less than letters
1. 1, C            in the ASCII standard
2. 2, d
3. 3, 0
4. 5, 9
5. 8, 2
6. 9, 1
7. C, 3            UPPERCASE is less than lowercase
8. a, 8            in the ASCII standard
9. d, 5
```

Notes:

- You can most often use strcmp() as the argument for cmp().
- It is necessary to declare the function cmp() before the call to quick(). The required ASM code for the call is LXI H,cmp ! PUSH H. Look out for the incorrect LHLD cmp ! PUSH H.
- To make a compare for mixed upper and lower case string data, write a function called mycmp() and use it instead of strcmp() in the example below.
- This source file is very portable. Use it on any system with minimal C compiler.

Reference:

BYTE Oct-84, p 369
By William M. Raike
B*Y*T*E J*A*P*A*N

The rad Function

Purpose:

Changes float x degrees to radians.

Function Header:

```
float rad(x)        Radian value of argument x.
float x;            Float value in degrees.
```

Returns:

angle in radians (a FLOAT)

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sin(),rad();
     printf("Enter float degrees x: "); gets(s); x = atof(s);
     printf("sin(rad(%f)) = %f, rad(%f) = %f, errno = %d\n",
            x,sin(rad(x)),x,rad(x),errno);
}
```

Notes:

- Method used is angle = (PI/180.0)*x.
- No error codes returned. No need to code for errno.
- No overflow check.
- No auto conversion to FLOAT.

The rand and srand Functions

Purpose:

Rand() generates pseudo-random numbers. Default seed is 2168. To get other seeds you must call srand().

Function Header:

```
unsigned rand()          Returns next random number in sequence.

VOID srand(seed)         Seeds the random number generator.
int seed;                Seed must be a 16-bit integer.
```

Returns:

Unsigned integer in the range 0 to 65535.

Example: Seed the random number generator from the Z100 2ms clock, then print 10 random numbers.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i,peekw(),rand();

     srand(peekw(11));
     for(i=0;i<10;++i) printf("%u\n",rand());
}
```

Notes:

- Srand(seed) initializes the pseudorandom number generator with the value of seed. The same seed produces the same series of numbers from function rand().
- The default value of 2168 is used in case you fail to seed the generator. This makes for rapid testing of programs. The seeding method is left unresolved until it becomes important.
- Older versions of the library included a function called rand1(). It still exists, for those cases when you want to have a large number of random number generators running simultaneously. See the source archives.

The rdDISK Function

Purpose:

Direct-disk read function for CP/M 2.2.

Function Header:

```
rdDISK(track,record,drive,buffer)

int track;              Physical track to read, 0 = first.
int record;             Record to read from track, 1 = first.
int drive;              Drive number. Use 0 for A:, 1 for B:, etc.
char buffer[128];       Location to copy disk data.
```

Returns:

Buffer filled on success.
Error code for a disk read is returned. Should be 0 for success.

Example: Read track 0 sector 1, the Zenith label sector.

```
#include <stdio.h>
main()
{
int i; char buf[128];
#define TRK 0
#define SEC 1
#define DSK 0
     printf("rdDISK() = %d\n", i = rdDISK(TRK,SEC,DSK,buf));
     for(i=0;i<128;++i) printf("%02x\n",buf[i]&255);
}
```

Notes:

- This was used to code the CROSS-CHECK source for the Z90. See Dr.Dobbs Journal, September, 1983. It is also useful for writing your own disk patcher or disk scanner.

- Note that rdDISK() is coded in assembly language and is highly non-portable. The function rdDISK() can be written in terms of the function bios(). This is recommended for portability, especially to CP/M- 68K and Unix. Here's the portable source:

```
bios(12,buffer);                Under CP/M-68K the second
bios(9,drive);                  argument of bios() has a
bios(10,track);                 cast of (long).
bios(11,record);                Under CP/M-80 2.2, the cast
return bios(13,0);              is (int).
```

- Use bios(13,1) to do an immediate read that gets around any LRU buffering scheme. Note that rdDISK() uses bios(13,0).

The read_ Function

Purpose:

Reads a block of characters off the disk in binary mode. The block must be a multiple of 128 bytes. Adheres to the Software Toolworks standard for read(). Use UNIX.H to obtain the Bell Labs Unix standard.

Function Header:

```
#include <stdio.h>
unsigned read_(fp,buffer,count)
FILE *fp;                       Open stream pointer.
char *buffer;                   Buffer for the read operation.
unsigned count;                 Number of bytes to read.
                                Must be a multiple of 128.
```

Returns:

The number of bytes actually read.
The number 0 is returned if EOF is reached.

Example: Read 2048 bytes from a disk file.

```
#include <stdio.h>
get2048(fp,buf)
FILE *fp;
char buf[2048];
{
unsigned x,read_();
     x = read_(fp,buf,2048);
     if(x != 2048) puts("Read less than 2048");
     if(x == 0) puts("Probably EOF");
}
```

Notes:

- Streams with descriptors 252,253,254,255 are devices. The C/80 read_() function cannot use these device descriptors. Use getc() for safety.

- Some systems return -1 on error. Your source code should always check read_() <= 0 rather than read_() == 0.

- A read failure is usually due to end of file. Since the C/80 read_() function operates in binary mode only, the ctrl-Z marker at the end of an ASCII file is not considered.

- The read_() function under C/80 operates in quanta of 128 bytes. You cannot read 1 character. See UNIX.H and the Unix-style read() and fread() functions for an alternative.

- This read_ function uses CP/M function 33 with manual advance of the record number. The initial record number is saved in the system variable IOsect[fp].

- The symbol **read_** is the same as **read** in CLIB.REL. Portable source code should use read_ instead of read. Old C/80 code should compile without changes. Unix code should use UNIX.H.

The read, reada Functions

Purpose:

Transfers bytes from a non-stream file to main memory. Move n bytes per call to a pre-assigned buffer. The transfer suffers from cr/lf translation if the file was opened in Ascii mode (the usual case).

Function Header:

```
int  reada(fd,buffer,n)
int fd;                 File descriptor, fd=fileno(fp) where
                        fp is the stream pointer.
char *buffer;           Pointer to base of memory storage
                        large enough to accept byte transfer.
int n;                  Number of bytes to transfer, 0...32767.
```

Returns:

```
m       The number of bytes actually transferred, an
        integer quantity 0...32767.
-1      Error
```

WARNING: In UNIX.H appears the definition

```
#define read reada
```

Example: Read 1 byte from the standard input.

```
#include <unix.h>
#include <stdio.h>
main()
{
int x;
     read(fileno(stdin),&x,1);
     fprintf(stderr,"Byte read = %d ASCII\n",x);
}
```

Notes:

- Useful for reading in Ascii data from the current file pointer.
- Recognizes devices "CON:", "RDR:".
- The primitive read() is used implicitly, but all Unix redirection is in force, because of the explicit use of getc().
- The UNIX.H definition of read() is reada(). It does correct I/O but suffers a slowdown with small buffer sizes.

The readb Function

Purpose:

Transfers bytes from a non-stream file to main memory. Move n bytes per call to a pre-assigned buffer. All byte transfers are in binary mode- no cr/lf translation.

Function Header:

```
int  readb(fd,buffer,n)
int fd;                 File descriptor, fd=fileno(fp) where
                        fp is the stream pointer.
char *buffer;           Pointer to base of memory storage
                        large enough to accept byte transfer.
int n;                  Number of bytes to transfer, 0...32767.
```

Returns:

```
m       The number of bytes actually transferred, an
        integer quantity 0...32767.
-1      Error
```

WARNING: In UNIX.H appears the definition

```
#define read readа
```

Example: Read one byte, binary mode, from a file entered on the command line. If it works, then print the byte.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv)
int argc; char **argv;
{
FILE *fp,*fopenb();
int x;
     if(argc > 1) {
       fp = fopenb(argv[1],"r");
       if(fp) {
          read(fileno(fp),&x,1);
          fprintf(stderr,"Byte read = %u\n",x);
     }
}
```

Notes:

- Useful for reading in binary data. All I/O is stream type. This function uses the library primitive getcbinary().
- The UNIX.H definition of read() is reada(), which does Ascii I/O correctly on files and devices.
- To get no-echo input, the best procedure is to write a special function like getbyte() that will at least cause the ported code to encapsule the problem. This feature is available on all systems but is indeed sometimes difficult to interface.
- Text editing features, ctrl-B processing and end of file detection other than end of record are all disabled under readb() when using the console device.

The realloc Function

Purpose:

Re-allocates contiguous memory located between the end of the program and the stack, copying the contents to the new region as required. Applies only to regions assigned by malloc().

Function Header:

```
char *realloc(ptr,s)
char *ptr;              Pointer to area already assigned
                        by malloc().
unsigned s;             New size of area.
```

The contents at ptr are preserved during the reallocation. If the new size s is smaller than the old size, then the extra contents are lost. Otherwise, new space is added on the end (not initialized), which will result in a memory move operation (a bit slow sometimes) which copies the contents.

Returns:

The base address of the area, on success.
0 if the request fails.

Example: Re-allocate a buffer area.

```
#include <unix.h>
#include <stdio.h>
main()
{
char *p,*malloc(),*realloc();

     p = malloc(2048);
     if(p == (char *)0) exit();
     printf("malloc(2048) = %x\n",p);
     for(i=0;i<10;++i) strcat(p,"This is a test of realloc\n");
     if(p = realloc(p,260)) {
       printf("realloc base = %x\n",p);
       for(i=0;i<260;++i) putchar(*p++);
     }
}
```

Notes:

- Uses malloc() to get raw system memory.
- You can't ask for more than 65531 bytes under CP/M-80.
- Sfree() doesn't know about realloc().
- Brk() doesn't know about realloc().

Structure: The structure used by malloc(), which is 4 bytes in length, precedes each block assigned by malloc(). Here is the definition:

```
struct block {
        struct block
             *nxtblk;
        unsigned
                 siz;
};
```

The rename Function

Purpose:

Change the name of an existing disk file.

Function Header:

```
int rename(oldname,newname)
char *oldname;                  Old file name.
char *newname;                  New file name.
```

Returns:

```
 0          No error.
-1          Error, rename operation failed.
```

Example: Rename a disk file.

```
#include <stdio.h>
main()
{
int rename();
char *findFIRST();
char s[129],t[129];
        printf("File name to change: ");
        gets(s);
        if(findFIRST('\0',s) == (char *)-1) {
                puts("File not found"); exit();
        }
        printf("New name: ");
        gets(t);
        if(rename(s,t) == -1) puts("Rename failed");
}
```

Notes:

- This function is found in most libraries but seems to be non-standard. Amazingly enough, the argument order is consistent among the major compilers.
- The related function unlink() which deletes a disk file is a Bell Labs standard function.

The reset Function

Purpose:

Resets the disk system under CP/M.

Function Header:

```
VOID reset()
```

Returns:

Nothing useful. Drive A: is logged-in, then the default drive.
All other drives are left in an unlogged state.

Example: Let the user change disks.

```
#include <stdio.h>
dskreset(drive)
char *drive;
{
char s[129];
  printf("Insert a new disk in %s and hit RETURN: ",drive);
  gets(s);
  reset();
}
```

Notes:

- Reset() is not a standard function. It is handy for writing interactive CP/M software.

- To simulate reset() on other systems, use the bdos() function or the ccBDOS() function.

```
#include <unix.h>
#include <stdio.h>
reset()
{
   return bdos(13,0);
}
```

```
#include <stdio.h>
reset()
{
   return ccBDOS(13);
}
```

The reverse Function

Purpose:

Reverses a string in place, e.g., "abcd" changed to "dcba".

Function Header:

```
char *reverse(str)
char *str;                    Source string.
```

Returns:

```
str                           Same base address, string reversed.
```

Example: Reverse a string of digits obtained by division.

```
#include <stdio.h>
main()
{
char s[129];
int i,n;
     printf("Enter a number: ");
     gets(s);
     n = atoi(s);
     i = 0;
     while(n>0) {
          s[i] = '0' + (n%10);
          ++i; n = n/10;
     }
     s[i] = '\0';
     printf("Number n before reversal: %s\n",s);
     reverse(s);
     printf("Number n after reversal: %s\n",s);
}
```

Notes:

- This function is described in K & R.
- Other uses of reverse() include swapping the string order in order to use strcpy() to copy a string into a circular buffer (in reverse order).
- An interesting exercise is to employ getline() to get a string, then reverse it and use ungetc() to put it back into the console buffer. Follow all this by getline() to see if its the same.

The rewind Function

Purpose:

Positions an open stream to the beginning of the file.

Function Header:

```
#include <stdio.h>        VOID and FILE defined herein.
VOID rewind(fp)
FILE *fp;                 Open stream pointer.
```

Returns:

Nothing.

Example: Rewind a file before changing the file buffer location. Access the file with a new in-memory buffer of user design.

```
#define CPMEOF 26
#define bufsz1 2048
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char buf[bufsz1];
FILE *fp,*fopen();
int i;
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("File not found"); exit(0);
     }
     if(getc(fp) == EOF) {
          puts("File empty"); exit(0);
     }
     rewind(fp);
     setbuf(fp,buf);
     getc(fp);
     for(i=0;i<bufsz1;++i) {
          if(buf[i] == CPMEOF) break;
          putchar(buf[i]);
     }
}
```

Notes:

- Rewind() has less overhead than fseek() or the older seek(). It does very little other than reset data areas in the FCB. No disk action takes place until I/O occurs.

- Setbuf() is supposed to include a call to rewind(). The above code is especially careful, because ported code may end up with a call to a setbuf() that does not call rewind.

- The size of a file buffer is fixed at compile time with most compilers. This library is unusual, in that the buffer size can be changed on the fly. However, it should always be preceded by a call to rewind(), to insure past data is on the disk and I/O starts anew.

- The companion function seekend() is useful to position the file pointer to the end of the file.

- While rewind is usually available on target systems, it generally calls fseek(). Note that seekend() is not expected to be on other systems. Both can easily be defined as a **code macro** in terms of fseek:

```
#define rewind(fp)  fseek(fp,0L,0)
#define seekend(fp) fseek(fp,0L,2)
```

The rindex Function

Purpose:

Finds the character position of a byte inside a string, starting from the end of the string.

Function Header:

```
char *rindex(str,c)
char *str;                  Source string.
char c;                     Character to locate.
```

Returns:

```
&str[i]                 Where c == str[i], on success,
                        where i =strlen(str) is decremented
                        to 0, and the first match is reported.
(char *)0               Failure.
```

Example: Find the optional extension in a file name and print.

```
#include <stdio.h>
main()
{
char *rindex();
char s[129],t[129],*p;
     printf("Enter a file name: ");
     gets(s);
     cvupper(s);
     if(p = rindex(s,'.')) strcpy(t,p+1);
     else strcpy(t,"");
     printf("File name extension: %s\n",t);
}
```

Notes:

- Usage of this function differs among libraries. The Unix V7 standard, according to Bourne, is the above.
- This function should be capable of finding the terminating null in a C string.
- Rindex() is the same function as **strrchr()**. See Harbison & Steele.

The run Function

Purpose:

Runs a CP/M 2.2 command line directly from a C program. The current program is lost and replaced by the new one. All CP/M command line arguments are sent to the new program.

Function Header:

```
VOID run(cmd)
char *cmd;              Program name (.COM assumed) plus the
                        command line tail.
```

Returns:

Nothing. The routine run() returns to the caller only in case the file is not found. Otherwise, the COM file is loaded and the the command line is passed.

Example: Run MBASIC.COM from a C program with a given command line.

```
#include <stdio.h>
main()
{
     run("MBASIC DUNGEON");
     puts("MBASIC not found");
}
```

Notes:

- Multiple arguments may be used. The command line limit of 128 characters must be obeyed, however. The .COM extension should not be used - it is assumed.
- All characters are acceptable, however uppercase translation is performed to simulate CP/M handling of command lines.
- The string may not contain a carriage return or linefeed.
- The string tail is decoded into the two CP/M default buffers and in addition it is copied to the CP/M default buffer at 0x80. The only real difference that CP/M might see is the lack of a command tail at the CCP. This problem has no easy solution, since the CCP is overlayed by the running program.

The sbinary Function

Purpose:

Binary search of a sorted string array to match a key string.

Function Header:

```
int sbinary(x,v,n)
char *x;                Key string to find in the array v[].
char *v[];              Sorted string array for lookup.
int n;                  Dimension of the array v[].
```

Returns:

```
-1      Failure. The test (strcmp(v[k],x)==0) failed for 0 <= k < n.
k       Success. A value of TRUE was found for (strcmp(v[k],x)==0).
```

Example: Look up a terminal in a sorted table.

```
static char *v[] = {
  "ADDS", "H19", "QUME", "TELERAY", "TELEVIDEO", "VISUAL200",
  "VISUAL500", "VT52", "WYSE", "Z29"
};
#include <stdio.h>
main()
{
int k;
char x[129];
     printf("Enter terminal name: ");
     gets(x); cvupper(x);
     k = sbinary(x,v,10);
     if(k == -1) puts("Not found");
     else
     printf("Found at index %d: %s\n",k,v[k]);
}
```

Notes:

- The assumption that the array v[] is SORTED is essential. It will probably hang the machine if the array is not sorted.
- A normal application uses sorted data entered in the program by hand as initializers for array v[].
- To obtain a sorted array on the fly use quick() or shell().

The sbrk Function

Purpose:

Obtains contiguous memory from the system and sets the upper bound of the program+data area.

Function Header:

```
char *sbrk(n)
int n;              Amount of memory requested. The maximum value
                    of n is 32767. See notes below.
```

Returns:

```
addr                on success, addr = base address of the
                    contiguous area of memory n bytes in length.
(char *)-1          on failure
```

Example: Get a buffer of 16k for special I/O.

```
#include <stdio.h>
main()
{
char *buf,*sbrk();
     if((buf = sbrk(16*1024)) == (char *)-1) {
          puts("Out of memory"); exit();
     }
     puts("It worked");
}
```

Example: Get all available memory for a buffer.

```
#include <stdio.h>
main()
{
char *buf,*sbrk();
unsigned memsize();
     if((buf = sbrk(memsize())) == (char *)-1) {
          puts("Out of memory"); exit();
     }
     puts("It worked");
}
```

Notes:

- For UNIX compatibility, restrain n to 32767 and call sbrk() more than once to get the desired amount of memory. This function will work with n an unsigned integer, at the expense of non-portable code.

- sbrk() checks for text, data and stack over-write. It also looks for attempts to write over file buffers and FCB areas.

- The return of -1 for failure is normal, but some libraries return 0. Beware as you attempt to port code to new machines.

- The return of sbrk(0) is the current program break. It does not change pointers - treat it as a report with no action.

- The symbols end, etext, edata have these meanings:

end	The next usable address after program load.
etext	The physical end of the program text before the file buffers and FCB buffers begin.
edata	Same as etext for C/80 because code and data are in the same place.

- The externals end, etext, edata are defined in the assembly module ETEXT.C as follows:

```
end:   DW  $END##+TOT_SZ##
edata: DW  $END##
etext: DW  $END##
```

- The double-# marks a symbol as external for M80, e.g, $END## is equivalent to EXTRN $END.

- The object code uses addresses 100h to $END, that is, the length of the compiled program is (etext-0x100).

- TOT_SZ is the total size of the file buffer and file control block (FCB) area that immediately follows the object code. See STDIO.H. The base address of the area is end (= $END). The area uses TOT_SZ bytes with last address (etext - 1).

- Reference: For end, edata, etext, see Banahan & Rutter, The Unix Book, Wiley Press (1984), p 198. For a description of brk() and sbrk(), see Banahan p 205.

The scanf Function

Purpose:

Parses formatted input text from the stream stdin using a control string to guide the conversion and storage of data items.

Function Header:

```
#include <stdio.h>
int scanf(control,&arg1,&arg2,...)
char *control;                    Control string
&arg1,&arg2,...                   Conversion storage locations.
                                  Pointers do not use the & prefix.
```

The control string is a template for the expected form of the input stream. The function of scanf() is to perform a simple match between the input stream and the control string. Contents of the control will be interpreted during processing as follows:

1. WHITE SPACE. Any white space in the control string causes an indeterminant amount of white space to be skipped in the input. The actual characters skipped will be: SPACE, TAB, CR, LF.

2. NON-CONVERSIONS. Characters other than white space must be matched by the input stream, except for %. The latter will match % if entered in the control string as %%. Otherwise, the % sign begins a block of conversion specifications, and the matching requirement is suspended until the conversion is processed.

3. CONVERSIONS. The allowed conversions parallel printf() usage. Each must start with %, optionally followed by decimal digits which control the maximum width of the conversion, then one, two or three conversion characters:

c	Character, white space too.
s	String, white space delimited, null appended.
[	String, [...] defines acceptable characters.
d	Decimal base 10, signed.
u	Decimal base 10, unsigned.
o	Octal base 8, unsigned.
x	Hexadecimal base 16, unsigned.
b	Binary base 2, unsigned.
e,E	Floating point, signed.
f	Same as e.
g,G	Same as e.
%	Match percent sign.
*	Convert item but do not store result.
ld	Long decimal base 10, signed.
lu	Long decimal base 10, unsigned.
lo	Long octal base 8, unsigned.
lx	Long hexadecimal base 16, unsigned.
lb	Long binary base 2, unsigned.

For example, "%12*ld" means to process at most 12 decimal digits for storage in a long integer location, but skip the storage. The effect is to read over incoming data.

In contrast, "%12ld" does the exact same conversion, but stores the answer at one of the pointers given in the argument list.

The conversion "%[A-Z,a-z]" reads in a string of characters until the character class [A-Z,a-z] is violated. The string is null-delimited after processing stops. Similarly, "%[^A-Z]" negates the character class [A-Z], and causes the read to continue until a character belongs to the class [A-Z]. The most common usage is "[^\r\n]", which is equivalent to the function gets() or LINE INPUT in BASIC.

All floating-point conversions are in single precision, since this library does not have double precision variables.

Long integer conversions are the same as short integer conversions, save the actual size of the final storage location. Short integers are 16 bits and long integers are 32 bits. It is up to the programmer to insure that the control string specifications match the storage size requirements of the argument list.

Returns:

(scanf(...)) is the return value, NOT scanf(). The extra parentheses are required to capture the scanf() return value.

The returned value is EOF if no items were read and EOF was encountered on input.

The returned value is the number of successfully processed items in all other cases. This count should match the number of arguments following the control string.

Example: Read in floating-point numbers from the console.

```
#define pfFLOAT     1
#define sfFLOAT     1
#include <stdio.h>
main()
{ char s[129]; float f;
     do {
          printf("Enter a float: ");
          i = (scanf("%f",&f));
          if(i)
            printf("%10.8f\n",f);
          else {
            printf("Bad input.\nPress RETURN: ");
            gets(s);
          }
     } while(i);
}
```

Example: Read in a string of characters and decode as hexadecimal.

```
#include <stdio.h>
main()
{
char s[129];
int x,i;
   do {
        puts("Enter hex digits");
        i = (scanf("%[^\r\n]",s));
        if((sscanf(s,"%x",&x)) > 0)
          printf("Hex value = %010x, Dec value = %d\n",x,x);
        else
          puts("Bad hex digits");
   } while(i);
}
```

Example: Read in three long integers per line, with comma delimiters.

```
#define pfLONG      1
#define sfLONG      1
#include <stdio.h>
main()
{
char s[129];
int i,j;
long x,y,z;

  do {
       puts("Enter 3 integers separated by commas on one line");
       i = (scanf("%[^\r\n]",s));
       j = (sscanf(s,"%ld, %ld, %ld",&x,&y,&z));
       if(j == 3)
         printf("x = %ld, y = %ld, z = %ld\n",x,y,z);
       else
         puts("Bad input - need 3 integers and 2 commas");
  } while(i);
}
```

Notes:

- Scanf() calls ungetc(x,fp) to put back the last character, when necessary. You can depend on scanf() to eat white space, in particular newlines.

- To use long and float features of scanf, write your code as follows:

```
#define sfFLOAT 1        /* include scanf float */
#define sfLONG  1        /* include scanf long  */
#include <stdio.h>
```

  Since the header file processes the sfFLOAT and sfLONG defines, the order is important. The default is to leave off the long and float code for an object code size savings of several kilobytes.

- The extra parentheses are required under C/80 because of the multiple argument kludge. Failure to include the parentheses will result in a stack address return rather than the number of items processed.

- It is often safer to use gets() or fgets() to input a line of text and then apply sscanf() to decode the string. The use of ungetc() and full stream I/O sometimes gets in your way, presenting surprises for bad input or end of file conditions.

- Scanf() often uses basic subroutines that gobble up character classes from the input. Generally, you have to write your own t get it right. Here's an example of how to advance past decimal digits in a string buffer:

```
char *digitpurge(s)
char *s;
{
int x;
     while((x = *s) != EOS) {
          if(isspace(x) || isdigit(x)) ++s;
          else break;
     }
     return s;
}
```

The seek Function

Purpose:

Positions the file pointer for an open stream.

Function Header:

```
int seek(fp,offset,type)
FILE *fp;                    Open stream pointer.
int offset;                  Offset from target position,
                             positive, negative or zero.
int type;                    Flag 0,1,2,3,4,5 determines
                             the target position and units:
```

type	Action taken for value 'offset'	Units
0	point to beginning + offset	bytes
1	point to current + offset	bytes
2	point to end + offset	bytes
3	point to beginning + offset	sectors
4	point to current + offset	sectors
5	point to end + offset	sectors

Returns:

0	Seek worked, file pointer reset as per request.
-1	Seek failed.

Example: Seek to the end of an existing file and append a message.

```
#include <stdio.h>
main()
{
FILE *fp,*fopen();
  if(argc <= 1) exit();
  fp = fopen(argv[1],"u");
  if(fp == (FILE *)0) {
    puts("Bad open"); exit();
  }
  if(seek(fp,0,2) == -1) {
    puts("Bad seek"); exit();
  }
  fprintf(fp,"Appended message\n");
  fclose(fp);
}
```

Notes:

- This set of primitives is used to build the Unix-style fseek() in CLIBU.REL. Use that function to generate code that ports easily to Bell Labs Unix V7.

- A seek of type 2 requires a file size computation, which under CP/M 2.203 is invalid unless the file control block is on disk. To insure the validity of the FCB, seek() writes the FCB back on the disk for a type 2 seek on a file opened for update or write.

- Binary files use sector fill of NULL (00 hex) while others use sector fill of CTRL-Z (1A hex). Carriage return and linefeed translation occur in files not opened in binary mode.

- Seeks on Ascii text files are not dependable (and the code is not portable). For example, moving forward one character may actually move two, because the first was a carriage return (0D hex).

- A rewind or seek to end of file is dependable and portable for ASCII or BINARY files. Likewise, a move by whole sector is Ok, but offsets into a sector can be off by the number of CR/LF pairs in that sector.

- **DO NOT USE the SEEK on the C/80 distribution disk.** It won't link (use that SEEK with Walt Bilofsky's CLIBRARY.REL).

The seekend Function

Purpose:

Seeks to the end of an open file given by descriptor fd. The C/80 file buffer pointer is positioned to the next byte to be written. This is a special C/80 function needed to make the Unix library perform as expected. Not intended to be portable.

Function Header:

```
seekend(fd)
int  fd;                File descriptor
```

Returns:

Nothing. It either works or CP/M will bounce you to the system.

Example: Seek to the end of a file in a ROM application.

```
#include <stdio.h>
myopen(file)
char *file;
{
int fd;

     fd = fopen (file,"r");        /* use C/80 fopen() */
     if(fd != -1) seekend(fd);     /* and unix seekend */
     return fd;
}
```

Notes:

- If the file was opened in binary mode, then the seek is to a record boundary.
- If the file was opened in Ascii text mode, then the buffer is filled from the last disk record and the file is positioned to the ctrl-Z at the end of the file or the next new record.
- This version is used in the Unix fopen() source to reduce the seek() overhead. It saves about 700 bytes in code size.
- The following externs from IOTABLE.CC are used:

```
extern  int
IOind[],IOsect[],IOfcb[],IOend[],IOsize[],IObuf[];
extern  char
IObin[];
```

- File descriptors are integer offsets into the above arrays. Here, stream pointers and file descriptors are identical.

The seldsk Function

Purpose:

Selects a disk drive under CP/M.

Function Header:

```
VOID seldsk(x)
int x;              Drive number 0...15 for drives A: to P:
```

Returns:

Nothing useful. The action taken is to mount the disk, read the file allocation bitmap to main memory and mark the disk as $R/W. All other drives are left untouched.

Example: Let the user change disks in drive C:.

```
#include <stdio.h>
main()
{
    log("C:");
}

log(drive)
char *drive;
{
char s[129];
  printf("Insert a new disk in %s and hit RETURN: ",drive);
  gets(s);
  reset();
  seldsk(toupper(drive[0])-'A');
}
```

Notes:

o Seldsk() is not a standard function. It is handy for writing interactive CP/M software.

o To simulate seldsk() on other systems, use the bdos() function or the ccBDOS() function.

```
#include <unix.h>
#include <stdio.h>
seldsk(x) int x;
{
  return bdos(14,x);
}
```

```
#include <stdio.h>
seldsk(x) int x;
{
  return ccBDOS(x,14);
}
```

The setatt Function

Purpose:

Set the CP/M file attributes, which are:

Read-only	$R/O or $R/W
System	$SYS or $DIR
Archive	Used by some backup programs, but not by CP/M at present.

Function Header:

```
int  setatt(name,code)
char *name;             File to update, null-terminated string
int  code;              Coded attribute word, 0 <= n <= 7:

                        code&1 = Read-only attribute
                        code&2 = System attribute
                        code&4 = Archive attribute
```

Returns:

0	Attributes set as requested
-1	File not found or set failed due to write-protect tab or disk directory write error.

Example: Set the attributes of a file entered on the command line.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int n;
    if(argc <= 2) { puts("Usage: A>main filename code"); exit(); }
    n=atoi(argv[2]);
    if(n < 0 || n > 7) {
        puts("Bad code, range is 0 to 7"); exit();
    }
    if(setatt(argv[1],n)) {
        puts("File not found or disk write-protected"); exit();
    }
    printf("File %s set to %s,%s,%s\n", argv[1],
           (n&1 ? "$R/O":"$R/W"), (n&2 ? "$SYS":"$DIR"),
           (n&4 ? "Archive":"Non-Archive") );
}
```

Notes:

- The name is a normal C string with NULL delimiter.
- The coded word uses only bits 0,1,2.

The setbuf Function

Purpose:

Changes the location of the current file buffer to the given address.

Function Header:

```
int  setbuf(fp,buffer)
FILE *fp;              Stream pointer, open stream
char *buffer;          Base address of the data area
                       to be used for a file buffer
```

Example: Change buffer size to 4096 and change buffer to a new location for further I/O.

```
#include <stdio.h>
setbsize(n,fd)
int n,fd;
{
 extern int IOsize[];
 extern MAXCHN[];
        if(1 <= fd && fd < MAXCHN) IOsize[fd] = n;
        else exit();
}
main()
{
extern int IOsize[];
int x;
FILE *fp,*fopen();
char buf[4096];

    if(fp = fopen("TEST","r")) {
        setbsize(4096,fp);
        setbuf(buf,fp);
        while((x=getc(fp)) != EOF) putchar(x);
    }
}
```

Notes:

- Stream files always have buffers set up by a call to sbrk(). You can run out of memory by using large file buffers that are not de-allocatted after use.
- The buffer size is fixed to the value in IOsize[fp]. You may change it in C/80, but note that this creates non-Unix source code.
- Change the file buffer location to where the current data buffer is located.

The setjmp and longjmp Functions

Purpose:

Provides a non-local GOTO which allows a graceful and certain exit from deeply nested procedures. Used primarily for backing out out a sequence of function calls during which a fatal error was encountered.

Function Header:

```
#include <setjmp.h>
int setjmp(env)
jmp_buf env[1];          Environment save area. Definitions are
                         in SETJMP.H header file.

VOID longjmp(env,v)
jmp_buf env[1];          Environment save area. Definitions are
                         in SETJMP.H header file.
int v;                   Value for longjmp() to return.
```

Returns:

setjmp:	0	Normal return.
longjmp:	v	Restores registers from env[], loads CPU register with value v. Processor jumps to last instruction in SETJMP (a RET).

Explanation of the example: The program below is an infinite loop that accepts keyboard characters.

Pressing certain keys causes an invisible GOTO to parts of main(). In this sense, setjmp() makes an invisible program LABEL and longjmp() is a GOTO LABEL.

The difference between this method and using actual labels is that the GOTO may act from within deeply nested functions with target label in another module.

Example: Test program to observe effect of setjmp(), longjmp().

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf env1[1];
jmp_buf env2[1];

f(fp)
FILE *fp;
{ int x;
     putchar(':');
     x = getcbinary(fp); putchar('\n');
     switch(x) {
     case 'Z'-'@': longjmp(env1,10);
     case '\n'   : longjmp(env1,20);
     case 'C'-'@': longjmp(env1,30);
     case 'A'-'@': longjmp(env2,1);
     }
}

main() {
   puts("Setup for env1");
   switch(setjmp(env1)) {
     case 0: break;
     case 10: puts("EOF encountered on input");
     case 20: puts("Linefeed in input"); break;
     case 30: puts("Ctrl-C normal exit"); exit();
     default: puts("Undefined error");
   }
   puts("Setup for env2");
   switch(setjmp(env2)) {
     case 0: break;
     case 1: puts("ctrl-A struck"); break;
     default: puts("Undefined error");
   }
   puts("Special characters are ctrl-A, ctrl-C, ctrl-J, ctrl-Z");
   while(TRUE) f(stdin);
}
```

Notes:

- Note that **jmp_buf** varies among systems as does the storage class of the array. All this info is hidden in the SETJMP.H header file, whose exact contents varies between systems.

- The definition of **jmp_buf** below is made in SETJMP.H in order to present a portable interface across compilers. The size of the structure is set at 4 to protect the STACK POINTER and RETURN ADDRESS that were in force when setjmp() was called to load the environment array. Note that PSW, BC, DE, HL are all clobbered by setjmp(). Longjmp() loads register HL with value v and sets the stack pointer to its old state.

```
struct jmpbuffer { char *STACKP; char *RETADR;};
#define jmp_buf struct jmpbuffer
```

- Here is a common error:

WRONG WAY	RIGHT WAY
jmp_buf env;	jmp_buf env[1];
setjmp(env);	setjmp(env);

  The compiler will complain about the error, so there is not much chance of it sneaking by the programmer.

- Portability of the C/80 code is not changed by adding the [1]. On a system using typedef, the result is

```
struct jmpbuffer env[1][1];
```

  instead of

```
struct jmpbuffer env[1];
```

  These statements generate exactly the same addressing mode and storage requirements.

The sfree Function

Purpose:

Frees memory that has been obtained by calls to core() or sbrk().

Function Header:

```
unsigned sfree(n)
unsigned n;              Amount of memory to free. The maximum
                         value of n is 65535. See notes below.
```

Returns:

The amount freed, which is n if it worked as planned, 0 if it failed completely, or some unsigned integer in between, which represents the function's attempt to please the caller within the restraints placed by the location of file buffers and file control blocks.

Example: Free up all the space assigned by sbrk() and core() after the the program is finished, so that a re-start will not run out of memory.

```
#include <stdio.h>
main()
{
char *p,*core();
     p = core(32767);
     fillchr(p,'A',32767);
     sfree(-1);
     puts("It worked");
}
```

Notes:

- File buffers and file control block areas will not be freed by sfree().
- The dynamic building of file buffers and file control blocks may leave large areas of memory unusable by sbrk() or sfree(). Avoid this problem by allocating enough built-in file buffers for the program to run without having to call sbrk() for more space.
- This function will remove orphan buffers from the heap if followed by a call to sfree:

```
unbuff(fd) int fd;
{
extern char IOch[]; extern int IOfcb[],IObuf[];
   if(IOch[fd] == -1) IOfcb[fd] = IObuf[fd] = 0;
}
```

The shellsort Function

Purpose:

Shellsort for an array of character strings with easy-to-change comparison method. Changes pointers but not actual data in the string.

Function Header:

```
shells(p,n,CMP)
char *p[];              Base address of pointer array
int  n;                 Number of array elements to sort
int  (*CMP)();          Compare routine for two strings p,q
                        that returns an integer with the
                        same rules used by Unix strcmp(p,q):

                           = 0     strings p,q are equal
                           < 0     p < q
                           > 0     p > q
```

Returns:

Nothing of value

Example: Sort an array of strings.

```
char *p[10] = {"a","C","d","0","9","2","1","3","8","5"};
char *q[10] = {"a","C","d","0","9","2","1","3","8","5"};
int strcmp();
#include <unix.h>
#include <stdio.h>
main()
{
int i;
       shells(p,10,strcmp);
       for(i=0;i<10;++i) printf("%d. %s, %s\n",i,p[i],q[i]);
}
```

Results from the above program

```
0. 0, a             Numerals are less than letters
1. 1, C             in the ASCII standard
2. 2, d
3. 3, 0
4. 5, 9
5. 8, 2
6. 9, 1
7. C, 3             UPPERCASE is less than lowercase
8. a, 8             in the ASCII standard
9. d, 5
```

Notes:

- You can most often use strcmp() as the argument for cmp().
- It is necessary to declare the function cmp() before the call to shells(). The required ASM code for the call is LXI H,cmp ! PUSH H. Look out for the incorrect LHLD cmp ! PUSH H.
- To make a compare for mixed upper and lower case string data, write a function called mycmp() and use it instead of strcmp() in the example above.
- This source file is very portable. Use it on any system with minimal C compiler.

Reference:

The C Programming Language, p 116,
by Kernighan & Ritchie
Prentice-Hall, Englewood Cliffs (1978)

The signal Function

Purpose:

Connects a C function with CPU interrupt processing. Currently a CP/M error return that allows code using Signal() to compile with no errors.

Function Header:

```
int (*signal (sig,f))()
int  sig;              Exception number to trigger signal().
int  (*f)();           C function to call when exception
                       number sig is encountered by the
                       interrupt system.
```

Returns:

```
 0              No error
-1              Argument sig not in range (present default).
```

Library routines preserve registers and condition codes. Function f() exits to the interrupted C process and normal program execution resumes.

Notes:

- Double translation of the number sig is required in order for signal() to set up an interrupt.
- At present, no implementation of signal() has been made. It is an error return to allow Unix source code to compile without errors, even though it will probably not run.
- Signal is a stub function like the other Unix System III compatibility functions. See UNIXIO for a complete list and Banahan & Rutter, The Unix Book, Wiley Press(1983) for function.

The sin Function

Purpose:

Compute the sine of real float value x. The value x is assumed in radians (not degrees).

Function Header:

```
float sin(x)            Sine.
float x;                Float value for computation.
```

Returns:

```
number          Answer between -1 and 1
                No error checking for large x
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sin();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("sin(%f) = %f, errno = %d\n",x,sin(x),errno);
}
```

Notes:

- Use deg() and rad() for conversions.
- Method used is Rational approximation from Cheney & Hart. Coefficients from #3370, C&H.
- Cody & Waite p 137 had too many errors with the current float package. Went back to Cheney & Hart to get it right.
- The switch case below uses more code but runs about 32% faster than other methods. Use ASM to optimize.
- No error checks for large x. The max size of x is about 1440 degrees, which is 4 wraps of the unit circle (25.13 radians).
- No assembler optimization.

The sinh Function

Purpose:

Compute the hyperbolic sine of real float value x.

Function Header:

```
float sinh(x)       Hyperbolic sine (e^x - e^-x)/2.
float x;            Positive, negative or zero argument.
```

Returns:

```
number          x in range
INF             x too large positive (see notes).
-INF            x too large negative (see notes).
                errno = ERANGE returned to flag error.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv)  int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sinh();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("sinh(%f) = %f, errno = %d\n",x,sinh(x),errno);
}
```

Notes:

- Method used is sinh(x) = (exp(x) - exp(-x))/2.0.
- The value of EXPLARGE was found by solving the equation exp(x) = 1.0e39.

The sob Function

Purpose:

Skips over leading white space. Stands for Skip Over Blanks.

Function Header:

```
char *sob(s)
char *s;                    String, null-terminated.
```

Returns:

The address of the null terminator in the string, or the address of the first non-blank character. Blanks are defined by isspace().

Example: Find and print the first word in a line of text.

```
#include <stdio.h>
main()
{
char buf[129];
char *s,*sob();
     printf("Enter a line of text: ");
     gets(buf);
     s = sob(buf);
     while(*s) {
          if(isspace(*s)) break;
          putchar(*s++);
     }
     putchar('\n');
}
```

Notes:

- This function is almost always used in conjunction with fnb(), in order to isolate tokens in a line of text.
- High-overhead functions like scanf() can often be avoided in a portable way by using sob() and fnb().
- Both sob() and fnb() return the address of the null delimiter of the string in case the character class checking runs off the end of the string.
- The source code for sob() is:

```
char *sob(s) char *s;
{
  while(isspace(*s)) ++s; return s;
}
```

The sprintf Function

Purpose:

Outputs formatted data to a string in memory using a control string and an appropriate argument list of variable length.

Function Header:

```
sprintf(s,control,arg1,arg2,...);
char *s;                          String in memory.
char *control;                    control string, see below
arg1,arg2,...                     appropriate arguments,
                                  8, 16 or 32-bit data, see below
```

The control string requirements are the same as for printf. See the printf() rules.

Returns:

Nothing.

Example: Print a number in Decimal, Octal, Hex, Binary to a string.

```
#include <stdio.h>
main()
{
char s[200];
  int i;
  for(i=0;i<128;++i)
    sprintf(s,"%-3d  %03o  %02x  %016b\n",i,i,i,i);
}
```

Example: Use the denser sprintf in the preceding example.

```
#define lib1 CLIBM
#include <stdio.h>
#undef sprintf
#define sprintf prnt_1(),prnt_3
main()
{
char s[200]; int i;
  for(i=0;i<128;++i)
    sprintf(s,"%-3d  %03o  %02x  %016b\n",i,i,i,i);
}
```

Notes:

- In this library, sprintf() is supplied in two versions. The fast version lacks some of the features found in the denser version for longs and floats. Both versions support multiple arguments. Neither version is fully recursive.

The sqrt Function

Purpose:

Compute the square root of real float value x

Function Header:

```
float sqrt(x)
float x;            Positive or zero argument for the square root.
```

Returns:

```
number          x >= 0 returns the square root of x.
0.0             x < 0 error
                errno = EDOM returned to flag error.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float sqrt();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("sqrt(%f) = %f, errno = %d\n",x,sqrt(x),errno);
}
```

Notes:

o Newton's method as here, or use sqrt(x) = pow(x,.5);

o Method from C&H, 6.1.3, 6.1.7. Startup from table 0231, C&H.

The sscanf Function

Purpose:

Parses formatted input text from a string in memory and stores the answers into given memory locations.

Function Header:

```
#include <stdio.h>
int sscanf(s,control,&arg1,&arg2,...)
char *s;                                String to be parsed.
char *control;                          Control string, see below
&arg1,&arg2,...                         Appropriate arguments
                                        See below for rules.
```

Returns:

The number of successfully parsed arguments. An error occurred if the number returned does not match the number of arguments following the control string. C/80 requires that sscanf() be enclosed in parentheses in order to check the return value, i.e.,

```
                i = (sscanf(s,"%d",&x));        RIGHT WAY
rather than
                i = sscanf(fp,"%d",&x);         WRONG WAY
```

Example: The following reads numbers from the console until either a data file error occurs or end of file is reached. Only one number per line is allowed.

```
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
char *gets();
char s[129];
int x;
     while(gets(s) != (char *)0) {
        if((sscanf(s,"%d",&x))>0) printf("%d\n",x);
        else break;
     }
}
```

Notes:

- The scanf family of functions accepts addresses only for its argument list. To error-check coding, verify that each argument has the address operator & as a prefix, or else the argument is a pointer (hence already an address).

- The converted values are stored at the given addresses in order left to right. Skips in the control string do not have an argument so the argument count may not match the conversion count.

- Short counts may hang the run-time package. Overly abundant counts will leave variables unfilled at run time.

- Always check the return of sscanf() to see if matches the expected value. It is easy to program infinite loops using sscanf().

- C/80 and its multiple-argument kludge cause users to write parentheses around sscanf() in order to recover the returned value. For example

```
if((sscanf(s,"%d",&x)) > 0)
```

  will not work under C/80 with the extra parentheses removed.

- To turn on float or long libraries for use with sscanf(), use the appropriate switches:

```
#define sfLONG  1       /* turn on long sscanf */
#define sfFLOAT 1       /* turn on float sscanf */
```

  The compiled code will change in size according to how much of the float and long libraries are actually used.

The ssort2 Function

Purpose:

Shell-Metzner sort in assembler for strings of fixed length which reside in an array of string pointers.

Function Header:

```
VOID ssort2(size,table,reclen)
int size;                   Number of pointers to sort.
char *table[];              Table of string pointers.
unsigned reclen;            Length of each string;
```

Returns:

Nothing. Pointers are altered to reflect the new sorted order. The actual data in memory is unchanged.

Example: Sort a list of names of fixed length.

```
#include <stdio.h>
main()
{
int i;
static
char *names[] = {
     "DAN RATHER  ",
     "E.G.MARSHALL",
     "JIMMY CARTER",
     "DONALD KNUTH"
};

  ssort2(4,names,12);
  for(i=0;i<4;++i) puts(names[i]);
}
```

Notes:

- The order of the arguments is essential.
- A fixed record length is required. For variable-length records see shell() and quick() in CLIBU.REL or ssort3() below.

The ssort3 Function

Purpose:

Shell-Metzner sort in assembler for strings of variable length which reside in memory. Access is by a special structure that simulates the string storage in Microsoft MBASIC.

Function Header:

```
struct mstr {
     char length; char *string;
};

VOID ssort3(size,&table)
int size;                    Number of pointers to sort.
struct mstr table[];         Table of string pointers.
```

Returns:

Nothing. Pointers are altered to reflect the new sorted order. The actual data in memory is unchanged.

Example: Sort a list of names of variable length stored as MBASIC strings.

```
#include <stdio.h>
main()
{
int i;
struct mstr {
     char length; char *string;
};
static
struct mstr names[] = {
     {10,"DAN RATHER"  },
     {12,"E.G.MARSHALL"},
     {12,"JIMMY CARTER"},
     {12,"DONALD KNUTH"},
     { 9,"TOM JONES"   },
     { 8,"007 BOND"    }
};
  ssort3(6,&names);
  for(i=0;i<4;++i) puts(names[i].string);
}
```

Notes:

- The order of the arguments is essential.
- The algorithm is Shell-Metzner's sort, with the strings left in place in memory. The table is re-organized to sorted order on return.

The stdin, stdout, stderr Streams

Purpose:

Defines the file pointer for standard input, standard output and standard error messages. Full re-direction capabilities are supported. No pipes or filters.

Function Header:

The definitions appear in STDIO.H, by C convention.

```
#define stdin (FILE *)fin__()
#define stdout (FILE *)fout__()
#define stderr (FILE *)252
```

Returns:

The stream pointer for the indicated stream, allowing for re-direction. No re-direction is possible for the standard error stream. It is always the console.

Example: Check the standard input stream to see if indirection is in force.

```
#include <stdio.h>
main()
{
     if(stdin == (FILE *)0)
          fprintf(stderr,"Stdin = console\n");
     else
          fprintf(stderr,"Stdin is file descriptor %d\n",stdin);
}
```

Notes:

- This implementation of stdin, stdout and stderr is at the macro preprocessor level, therefore it generates no object code in the COM file. Linkage, in case of actual use, causes module FINFOUT.REL to be linked, which adds 8 bytes to the object code. The routines fin__() and fout__() simply return the stream descriptors fin and fout.

- The stream stderr is hard-wired to the console. You can change it by re-assembly of the C program, but not by command line options.

- The descriptors fin and fout are set to zero at the start of a program. Re-direction alters these descriptors. Note that fin or fout can be closed by the user. After doing so, the user should set fin = fout = 0.

The strcat Function

Purpose:

Appends a null-terminated string str2 to the null-terminated string str1.

Function Header:

```
char *strcat(str1,str2)
char *str1,*str2;            Source strings, null-terminated.
```

Returns:

str1 The base address of the destination string.

Example: Fix a CP/M file name.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[256];
char *index();
     if(argc <= 1) exit();
     strcpy(s,argv[1]);
     if(index(s,'.') == (char *)0) strcat(s,".COM");
     printf("s = %s\n",s);
}
```

Notes:

- Strcat() is dangerous. The classic error is to append a string to a character pointer that is not null-terminated.
- Storage over-run is possible. There is no run-time check for writing beyond the limits of storage.
- See also strcpy, strncpy(), strncat() and moveMEM().
- Here is the standard strcat() code:

```
char *strcat(s1,s2) char *s1,*s2;
{
char *p;
  p = s1;
  while(*s1) ++s1; while(*s1++ = *s2++) ; return p;
}
```

The strchr Function

Purpose:

Searches for character c inside string s.

Function Header:

```
char *strchr(s,c)
char *s,c;              Source string s and character c to find.
```

Returns:

```
(char *)0               no match
&s[i]                   address of first match to s[i] == c
&s[n]                   n=strlen(s), if c = '\0'
```

Example: Find the end of a C string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static
char *s = "This is a Test";
char *p;
char *strchr();
     p = strchr(s,'\0');
     printf("Search for null at end of a string s.\n");
     printf("s = %u, p = %u, strlen(s) = %u\n",s,p,strlen(s));
}
```

Notes:

o strchr() is the same as index().

o Here is the standard strchr code:

```
char *strchr(s,c) char *s,c;
{
   do {
      if(*s == c) return s;
   } while(*s++);
   return (char *)0;
}
```

The strcmp Function

Purpose:

Compares two null-terminated strings for equality, less than or greater than.

Function Header:

```
int strcmp(str1,str2)
char *str1,*str2;            Source strings, null-terminated.
```

Returns:

```
 < 0       for str1 < str2
 = 0       for str1 = str2 (same length and characters)
 > 0       for str1 > str2
```

Example: Test a string for a match to one on the four CP/M devices.

```
#include <stdio.h>
main()
{
int i,j; char s[129];
static char *device[] = { "CON:","LST:","RDR:","PUN:" };
     printf("Enter a device name: ");
     gets(s); cvupper(s);
     for(i=0;i<4;++i) {
          if(strcmp(device[i],s)==0) {
               printf("Device match is: %s\n",device[i]);
               printf("Mismatches:\n");
               for(j=0;j<4;++j) {
                    if(j != i) puts(device[j]);
               }
               return;
          }
     }
     puts("No match");
}
```

Notes:

- Strcmp() does what is expected when one or both of the strings are null strings. Other cases of interest: "abc" > "ABC", "abc" < "abcd".

- Standard strcmp() source code:

```
int strcmp(s1,s2) char *s1,*s2;
{
  while(*s1 == *s2) { if(*s1 == '\0') return 0; ++s1; ++s2; }
  return (*s1 - *s2);
}
```

The strcpy Function

Purpose:

Copies a null-terminated string str2 to the storage area given by the string str1. The destination string is also null-terminated.

Function Header:

```
char *strcpy(str1,str2)
char *str1;                    Destination string.
char *str2;                    Source string, null-terminated.
```

Returns:

str1 The base address of the destination string.

Example: Copy a command line argument to safety.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[256];
     if(argc <= 1) exit(); strcpy(s,argv[1]);
     printf("argv[1] = %s\n",s);
}
```

Example: Copy lines of a file into an array.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i,n;
FILE *fp,*fopen();
char buf[129],*s[256];
char *core(),*fgets();
     if(argc <= 1) {
          puts("Usage: A>load filename"); exit();
     }
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Bad open"); exit();
     }
     i = 0;
     while(i<256 && fgets(buf,128,fp) != (char *)0) {
          s[i] = core(1+strlen(buf));
          strcpy(s[i],buf);
          ++i;
     }
     fclose(fp);
     n = i;
     puts("The saved lines:");
     for(i=0;i<n;++i) printf("s[%d]: %s\n",i,s[i]);
}
```

Notes:

- Strcpy() is dangerous because there is no check for buffer limits.

- Besides storage over-run caused by insufficient estimate of the requirements, there is the problem on uninitialized data:

```
char *s;
strcpy(s,t);
```

  The problem here is that s is initialized to 0, the base of the operating system. The strcpy() code above writes string t to address 0. This may cause a CP/M system to crash. To fix the problem, use core() to get storage for s, or use an auto array:

```
char *s;                    char *s;
char buf[256];              char *core();
s = buf;                    s = core(256);
strcpy(s,t);                strcpy(s,t);
```

- See also strncpy() and moveMEM().

- Standard strcpy source code:

```
char *strcpy(s1,s2) char *s1,*s2;
{
char *p;
  p = s1; while(*s1++ = *s2++) ; return p;
}
```

The strcspn Function

Purpose:

Searches a null-terminated string s for the first match to any character in the null-terminated string called set.

Function Header:

```
int strcspn(s,set)
char *s;                Target string s to be searched.
char *set;              Possible characters to match.
```

Returns:

strlen(s)	No match occurred.
i	The first index i where s[i] matches a character of string set.

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If no character of s is in B, then return strlen(s), otherwise return the first index n for which s[n] is in B.

Example: Find the first digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789"; int i;
     if(argc > 1) {
       if((i = strcspn(argv[1],set)) < strlen(argv[1]))
         printf("Matched at offset %d\n",i);
     }
}
```

Notes:

o Standard strcspn source code:

```
int strcspn(s,set) char *s,*set;
{
int n; char *p;
  n = 0;
  while(s[n]) {
    p = set;
    while(*p) {
      if(*p++ == s[n]) return n;
    }
    ++n;
  }
  return n;
}
```

The strlen Function

Purpose:

Computes the integer length of a null-terminated string.

Function Header:

```
int strlen(str)
char *str;          Source string, null-terminated.
```

Returns:

```
0               String has no characters (empty string "").
n               String has n characters.
```

Example: Find the length of a user-entered token.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
     if(argc <= 1) exit();
     printf("argv[1] = %s, strlen(argv[1]) = %d\n",
             argv[1], strlen(argv[1]) );
}
```

Notes:

- strlen() returns an unsigned integer. If you get a negative value for strlen(), then suspect a program bug. It is quite unusual to have a C string in excess of 32767 bytes.

- Strlen() is often used to find the null position in a C string. If p is a string pointer, then p += strlen(p) computes a pointer to the null terminator of the string.

- Standard strlen source code:

```
int strlen(s)
char *s;
{
int n;
  n = 0;
  while(s[n]) ++n;
  return n;
}
```

The strncat Function

Purpose:

Appends a null-terminated string str2 to the storage area given by the string str1. At most n bytes are transferred. The destination string is ALWAYS null-terminated.

Function Header:

```
char *strncat(str1,str2,n)
char *str1,*str2;            Source strings, null-terminated.
int n;                       Number of bytes to transfer.
```

Returns:

str1 The base address of the destination string.

Example: Fix a CP/M file name.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[25]; char *index();
     if(argc <= 1) exit();
     if(p = index(s,':')) s[0] = '\0';
     else strcpy(s,"A:");
     strncat(s,argv[1],20); printf("s = %s\n",s);
}
```

Notes:

- Strncat() is dangerous. The classic error is to append a string to a character pointer that is not null-terminated.

- The following problem can occur with strncat:

```
char *s;
strncat(s,t,1+strlen(t));
```

  The problem here is that s is initialized to 0, so strncat() will look for the end of the string that starts at the operating system base 0. Surely it will be found, then strncat appends str2. The end result is a very unpredictable system crash.

- Standard strncat() source code:

```
char *strncat(s1,s2,n) char *s1,*s2; int n;
{ char *p;
  p = s1; while(*s1) ++s1;
  while((n > 0) && ((*s1++ = *s2++) != '\0')) --n;
  *s1 = '\0'; return p;
}
```

The strncmp Function

Purpose:

Compares two null-terminated strings for equality, less than or greater than. During comparison, at most n characters are considered.

Function Header:

```
int strncmp(str1,str2,n)
char *str1,*str2;            Source strings, null-terminated.
int n;                       Maximum comparison count.
```

Returns:

```
 < 0       for str1 < str2
 = 0       for str1 = str2 (same length and characters)
 > 0       for str1 > str2
```

Example: Test a string for a match to one on the four CP/M devices.

```
#include <stdio.h>
main()
{
int i,j; char s[129];
static
char *device[] = { "CON:","LST:","RDR:","PUN:" };
     printf("Enter a device name: ");
     gets(s); cvupper(s);
     for(i=0;i<4;++i) {
          if(strncmp(s,device[i],4)==0) {
               printf("Device match is: %s\n",device[i]);
               printf("Mismatches:\n");
               for(j=0;j<4;++j) {
                    if(j != i) puts(device[j]);
               }
               return;
          }
     }
     puts("No match");
}
```

Notes:

o Strncmp() is strcmp() with a restraint. Standard source code:

```
int strncmp(s1,s2,n) char *s1,*s2; int n;
{
  while((n-- > 0) && (*s1 == *s2)) {
    if(*s1 == '\0' || n == 0) return 0; ++s1; ++s2;
  }
  return (*s1 - *s2);
}
```

The strncpy Function

Purpose:

Copies a null-terminated string str2 to the storage area given by the string str1. Exactly n bytes will be transferred, with null fill used if str2 is less than n bytes in length.

Function Header:

```
char *strncpy(str1,str2,n)
char *str1,*str2;              Source strings, null-terminated.
int n;                         Number of bytes to transfer.
```

Returns:

str1 The base address of the destination string.

Example: Copy a command line argument to safety.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[11];
     if(argc > 1 && strlen(argv[1]) > 10) {
          printf("Bad token length"); exit();
     }
     strncpy(s,argv[1],1+strlen(argv[1]));
     printf("argv[1] = %s\n",s);
}
```

Notes:

- Strncpy() is dangerous. An error can be made in its usage which allows the destination string to be non-terminated (no null at the end). Strncpy() will not write a null terminator unless strlen(str2)<n.

- The following code will over-write the operating system because the initial value in s is 0:

```
char *s;
strncpy(s,t,1+strlen(t));
```

- Conversion of random file routines from Microsoft's MBASIC can use strncpy() to write a field in a random record with 0 fill.

- Standard strncpy() source code:

```
char *strncpy(s1,s2,n) char *s1,*s2; int n;
{ char *p; p = s1;
  while(n-- > 0) { *s1++ = ( (*s2) ? *s2++ : '\0'); } return p;
}
```

The strpbrk Function

Purpose:

Searches a null-terminated string s for the first match to any character in the null-terminated string called set.

Function Header:

```
char *strpbrk(s,set)
char *s;                Target string s to be searched.
char *set;              Possible characters to match.
```

Returns:

```
(char *)0       No match occurred.
&s[i]           Address of the the first match.
```

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If no character of s is in B, then return NULL, otherwise return the first pointer &s[n] for which s[n] is in B.

Example: Find the first digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789";
char *p,*strpbrk();
     if(argc > 1) {
       if(p = strpbrk(argv[1],set))
         printf("argv[1] = %u, match at %u\n",argv[1],p);
     }
}
```

Notes:

o Standard strpbrk source code:

```
char *strpbrk(s,set)
char *s,*set;
{
char *p;
  while(*s) {
    p = set;
    while(*p) {
      if(*p++ == *s) return s;
    }
    ++s;
  }
  return (char *)0;
}
```

The strpos Function

Purpose:

Scan null-terminated string s for a match to character c.

Function Header:

```
int strpos(s,c)
char *s;                Target string s to be searched.
char c;                 Character c to locate.
```

Returns:

```
-1                  failure
strlen(s)           if c == '\0'
n                   n is the first index with s[n] == c
```

Example: Find the first letter 'A' in a string.

```
#include <stdio.h>
main(argc,argv)  int argc; char **argv;
{
int i;
     if(argc > 1) {
       if((i = strpos(argv[1],'A')) != -1)
         printf("argv[1] == 'A' at offset %u\n",i);
     }
}
```

Notes:

o This is the integer offset version of the function index().

o Standard strpos source code:

```
int strpos(s,c)
char *s,c;
{
int n;
  n = 0;
  do {
       if(s[n] == c) return n;
  } while(s[n++] != '\0');
  return -1;
}
```

The strrchr Function

Purpose:

Search null-terminated string s for the last match to character c.

Function Header:

```
char *strrchr(s,c)
char *s;                Target string s to be searched.
char c;                 Character c to locate.
```

Returns:

```
(char *)0       No character of s matches c.
&s[i]           i is the largest index for which s[i] == c
```

Example: Find the last occurrence of letter 'A' in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char *p,*strrchr();
     if(argc > 1) {
       if((p = strrchr(argv[1],'A')) != (char *)0)
         printf("Last argv[1] == 'A' at address %u\n",p);
     }
}
```

Notes:

o This is the same function as rindex().

o Standard strrchr source code:

```
  char *strrchr(s,c)
  char *s,c;
  {
  int n;
    n = 0;
    while(s[n]) ++n;
    do {
         if(s[n] == c) return (&s[n]);
    } while(n--);
    return (char *)0;
  }
```

The strrpbrk Function

Purpose:

Search null-terminated string s for the last match to any character in the null-terminated string called set.

Function Header:

```
char *strrpbrk(s,set)
char *s;                Target string s to be searched.
char *set;              String of possible matching characters.
```

Returns:

```
(char *)0           no match occurred, otherwise
&s[i]               address of the LAST match
```

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If no character of s is in B, then return NULL, otherwise return the last pointer &s[n] for which s[n] is in B.

Example: Find the last occurrence of a digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789";
char *p,*strrpbrk();
  if(argc > 1) {
    if((p = strrpbrk(argv[1],set)) != (char *)0)
      printf("Last digit of argv[1] = %u at %u\n",argv[1],p);
  }
}
```

Notes:

o Standard strrpbrk source code:

```
  char *strrpbrk(s,set)
  char *s,*set;
  {
  char *p;
  int n;
    n = 0;
    while(s[n]) ++n;
    do {
        p = set;
        while(*p) { if(*p++ == s[n]) return (&s[n]); }
    } while(n--);
    return (char *)0;
 }
```

The strrpos Function

Purpose:

Reverse scan null-terminated string s for a match to character c.

Function Header:

```
int strrpos(s,c)
char *s;                Target string s to be searched.
char c;                 Character c to match.
```

Returns:

```
-1                  Failure.
 n                  n is the largest index with s[n] == c.
 strlen(s)          If c = '\0'.
```

Example: Find the last occurrence of character 'A' in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int i;
  if(argc > 1) {
    if((i = strrpos(argv[1],'A')) != -1)
      printf("Last argv[1] == 'A' at offset %d\n",i);
  }
}
```

Notes:

o Standard strrpos source code:

```
int strrpos(s,c)
char *s,c;
{
int n;
  n = 0;
  while(s[n]) ++n;
  do {
      if(s[n] == c) return n;
  } while(n--);
  return -1;
}
```

The strspn Function

Purpose:

Search null-terminated string s for the first character that is not in the string called set.

Function Header:

```
int strspn(s,set)
char *s;                Target string for testing.
char *set;              String of characters to skip over.
```

Returns:

```
strlen(s)       All chars in s are also in set.
n               n is the first index with s[n] not in set.
0               String set is a null string.
```

Set Theory: Let B = {set[j] : 0 <= j < strlen(set)}. If all characters of s are in B, then return strlen(s), otherwise return the smallest index n for which s[n] is not in B. (B is empty if set = "\0")

Example: Find the first non-digit in a string.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
static char *set = "0123456789"; int i;
  if(argc > 1) {
    if((i = strspn(argv[1],set)) != 0 && i < strlen(argv[1]))
      printf("First non-digit of argv[1] is at offset %d\n",i);
  }
}
```

Notes:

o Standard strspn source code:

```
int strspn(s,set) char *s,*set;
{
char *p; int n;
  n = 0;
  if(*set == '\0') return 0;
  while(s[n]) {
    p = set;
    while(*p) {
      if(*p++ == s[n]) break; if(*p == '\0') return n;
    }
    ++n;
  }
  return n;
}
```

The swab Function

Purpose:

Copies one area of memory to another, reversing pairs of bytes. The number to copy must be even.

Function Header:

```
int  swab(s,d,n)
char *s;                Char pointer to source data area
char *d;                Char pointer to destination data area
int  n;                 Integer number of bytes to copy
```

Returns:

0 Always.

Example: Sample program to swap halves of words in an array.

```
#include <unix.h>
#include <stdio.h>
char *p = "abababababababababababababababababababababab";
char *q = "ccccccccccccccccccccccccccccccccccccccccccc";
main(argc,argv)
int argc;
char *argv[];
{
int i;
        i = strlen(p);
        puts(p);
        puts(q);
        swab(p,q,i);
        puts(q);
}
```

Output from the above main program:

```
A>main
abababababababababababababababababababababab
ccccccccccccccccccccccccccccccccccccccccccc
babababababababababababababababababababababa
A>
```

Notes:

- Copies data from a reverse byte-sex machine. For example, 68000 CPU integer data is reversed from the Intel Reverse Format found on all Z80, 8080, 8088, 8086, 80186, 80286 processors.

The system Function

Purpose:

Pass a command string to the console command processor.

Function Header:

```
int  system(s)
char *s;                CP/M command lines to run, string is
                        null-terminated. See example below.
```

Returns:

```
Never           On success, exits to system
0               On failure
```

Example: To do the CP/M commands:

```
CC TEST
M80 =TEST
L80 TEST,TEST/N/E
ERA TEST.MAC
```

Place this this code in your C program:

```
system("CC TEST\nM80 =TEST\nL80 TEST,TEST/N/E\nERA TEST.MAC");
```

Internals: This program creates a **$$$**.SUB file on drive A:, erasing any existing **$$$**.SUB file. The lines present in the command string are copied to the **$$$**.SUB file, for execution in the typed order.

Notes:

- The command processor (CCP) is to be loaded automatically by CP/M on exit.
- To return to the calling program, the string passed to the CCP must contain a re-load for the current program.
- CP/M is not concurrent. There is no efficient way to save the current process for a restart.

Other ways to do the same thing:

- Use run(). It accepts a CP/M command line to load and run a COM file. Only one line.
- See also chain().

The tan Function

Purpose:

Compute the tangent of real float value x. The value x is assumed in radians (not degrees).

Function Header:

```
float tan(x)
float x;            Float value in range -8*PI to +8*PI,
                    PI = 3.14159.
```

Returns:

```
number          Answer between -INF and +INF, errno = 0.
INF             x approx (2n+1)PI/2 error, errno = ERANGE.
ZERO            Argument too large negative or positive
                ( greater than 8*PI), errno = EDOM.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float tan();
     errno = 0;
     printf("Tan(x) accepts x from -25.13 to +25.13 radians\n");
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("tan(%f) = %f, errno = %d\n",x,tan(x),errno);
}
```

Notes:

- Use deg() and rad() for conversions.

- The method used is sin(x)/cos(x) in range, or rational approximation, Cheney & Hart, in the critical range:

```
tan(x) = sin(x)/cos(x)   for x not within .01 of
                         a multiple of PI/2
tan(x) = new approx      otherwise
```

- Current problems:

  First, if fabs(x) > 25.13, then we probably can't compute it at all. The number 25.13 comes from

```
8*PI = 25.13274123 (12-digits)
8*3.14159 = 25.13272 (6-digits)
error = -2.12288e-05
So to maintain 6-digit accuracy restrains
computation to -8PI < x < 8PI.
```

  Let us assume 0 <= x <= pi/2. Let u = pi/2 - x. If u is too close to zero, then computation fails:

```
sin(x)/cos(x) = sin(x)/cos(pi/2-u)
              = sin(x)/sin(u)
              = cos(u)/sin(u)
              = 1./u approximately
```

  This is cured by using a different rational function near pi/2, namely a Taylor expansion:

$$\tan x = (1/u)(1 - u^2/2 + u^4/24)/(1 - u^2/6 + u^4/120)$$

  At u = .009999, tan x = 100.006668, but 1/u = 100.010001. For 0 <= x < BADTAN, sin(x)/cos(x) is a good approximation. It looks like we need the above rational, but perhaps a better choice of BADTAN would allow use of 1/u.

- We define BADTAN = 1.560796327, which is pi/2 - .01 or 89.427 degrees. This rather arbitrary value comes from comparison with other tangent functions on micros and mainframes.

The tanh Function

Purpose:

Compute the hyperbolic tangent of real float value x.

Function Header:

```
float tanh(x)       Hyperbolic tangent (e^x - e^-x)/(e^x +e^-x).
float x;            Positive, negative or zero argument.
```

Returns:

```
number          x in range
INF             x too large positive (see notes).
-INF            x too large negative (see notes).
                errno = ERANGE returned to flag error.
```

Example:

```
#define pfFLOAT 1
#include <math.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
float x;
extern int errno;
float atof();
float tanh();
     errno = 0;
     printf("Enter a float x: "); gets(s); x = atof(s);
     printf("tanh(%f) = %f, errno = %d\n",x,tanh(x),errno);
}
```

Notes:

o Method used is tanh(x) = (exp(2x) - 1)/(exp(2x) + 1).

o The value of EXPLARGE was found by solving the equation exp(x) = 1.0e39.

The timer Function

Purpose:

Time an event under CP/M with 2ms clock at address 11. Works under CP/M 2.203 H89 or CP/M-85 Z100. Adaptable to most systems.

Function Header:

```
VOID timer(n)
int  n;                 0 to start timer
                        1 to print out the elapsed time
```

Returns:

Nothing of interest

Example: Time an internal loop and report the elapsed time.

```
#include <stdio.h>
main()
{
int i;
     timer(0);
     for(i=0;i<30000;++i) {
     /* an empty loop. The Sieve Benchmark would be interesting */
     }
     timer(1);
}
```

The timer routine prints the elapsed time using integer arithmetic and printf. It is accurate to 1/10 of a second.

Notes:

- Uses a fixed CLOCK word at address 11. The expected 2ms clock is present only on the H89 under CP/M and the Z100 under CP/M-85.
- See the source in TIMER.C for other ideas.

The toascii Function

Purpose:

Converts an integer into the ASCII range 0 to 127 by stripping the parity bit.

Function Header:

```
int toascii(x)
int x;                    Integer to convert to ASCII range.
```

Returns:

The converted value, in the range 0 to 127 decimal.
Logically equivalent to (x & 127).

Example: Read a word processor file such as produced by MicroPro's Wordstar, strip the file to ASCII range, and print on the printer.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int x;
FILE *fp,*fo,*fopen();

     if(argc <= 1) {
          puts("Usage: A>strip inputfile"); exit();
     }
     if((fp = fopen(argv[1],"r")) == (FILE *)0) {
          puts("Bad open of input file"); exit();
     }
     if((fo = fopen("LST:","w")) == (FILE *)0) {
          puts("Printer failure"); exit();
     }
     while((x = getc(fp)) != EOF) {
          putc(toascii(x),fo);
     }
     fclose(fo); fclose(fp);
}
```

Notes:

- With IBM data using the EBCDIC standard character set, the ASCII standard toascii() is totally useless.
- Toascii() is an honest function and not a macro. There are no side effects.

The toint Function

Purpose:

Converts a hex digit into a integer in the range 0 to 15. For example, toint('A') = 10, toint('f') = 15, toint('0') = 0.

Function Header:

```
int toint(x)
char x;                 Character to convert
```

Returns:

n	In range 0...15 for a valid upper or lower case hexadecimal digit from the character set 0123456789ABCDEFabcdef
-1	Error, an invalid hex digit.

Example: Read and convert hex digits from stdin.

```
#include <unix.h>
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
int x,y; FILE *fp,*fopenb();
  if(argc > 1) {
  if(fp = fopenb(argv[1],"w")) {
    while((x = getchar()) != EOF) {
       if(isspace(x)) continue;
       if((y = getchar()) != EOF)
            putc(toint(x)*16+toint(y),fp);
    }
    fclose(fp);
  }
}
```

Notes:

o Implementations of C that use code macros often put the function toint() into the CTYPE.H file. Here is the standard toint() code:

```
int toint(x) char x;
{
  if(isdigit(x)) return (x - '0');
  x = toupper(x) - 'A' + 10;
  return (10 <= x && x <= 15) ? x : -1;
}
```

The tokens Function

Purpose:

Decodes a CP/M command line into token strings. Processes re-direction files < and >. Handles strings with imbedded white space with single or double quote delimiters.

Function Header:

```
int tokens(table,cmdtail)
char *table[];          Pointer table to be filled.
char *cmdtail;          Special buffer area. The first byte is
                        the length of the string. Following are
```

The token table is altered by the function call as follows:

table[0]: address of '<' redirection file name
table[1]: address of '>' redirection file name
table[2]: address of the first token
.
.
.
table[n]: address of the last token
table[n+1]: always set to -1, as a marker

Returns:

The number of tokens actually processed.

Example: Do our own token decoding in lowercase.

```
#include <stdio.h>
decodtokens(table)
char *table[];
{
int n,i;
char s[129];
     printf("Enter a line: ");
     gets(&s[1]);
     s[0] = strlen(&s[1]);
     table[0] = table[1] = "NO DATA";
     n = tokens(table,s);
     printf("table[0] = Re-direction <: %s\n",table[0]);
     printf("table[1] = Re-direction >: %s\n",table[1]);
     for(i=2;i<2+n;++i) printf("table[%d] = %s\n",table[i]);
}
```

Notes:

- The character pointer **source** points to a buffer area in memory which contains the command line tail. This area is disected into tokens, which are delimited by nulls, and the addresses are consecutively stored in the table.

- Any redirection commands cause the entries **table[0]** and **table[1]** to be updated, otherwise these memory locations are left in entry state. Programs which use tokens() must initialize table[0] and table[1] (to a null string).

- Decoding recognizes both double and single quotes as commands to stop white space searching (until the quote is repeated). Therefore, tokens may be passed which contain white space.

- Both upper and lower case characters are recognized by tokens(). However, the CCP will convert lowercase to uppercase. Programs which require lower case input can therefore invoke tokens() inside the program, and service command line strings as usual. **See also insert(), chain(), run().**

- The redirection file names are obtained from pointers table[0] and table[1]. This feature allows internal redirection changes, by closing streams **stdin** and **stdout**, then re-opening these streams with the supplied file names. Implement this by writing your own **Copen_()** and **Cexit_()**. See also UNIX.H and the function freopen().

The tolower Function

Purpose:

Converts a character in the range A-Z to lower case.

Function Header:

```
int tolower(x)
int x;                          Character to convert.
```

Returns:

The character x, if no conversion was required.
The lowercase equivalent of x, if 'A' <= x <= 'Z'.

Example: Convert an input line to all lowercase.

```
#include <stdio.h>
main()
{
char s[129];
int i;

     printf("Enter a line of text: ");
     gets(s);
     for(i=0;i<strlen(s);++i) s[i] = tolower(s[i]);
     printf("Lowercase line:\n%s\n",s);
}
```

Notes:

- On some machines this function is a macro. Beware of side effects when porting code. If you write it, then try to do so without introducing possible side effects (like tolower(*p++)).
- Tolower() is an honest function, it is not a macro, and it has no side effects.
- Some macro implementations do not bother testing to see if the integer x to be converted is in the range A-Z. The net result is an invisible bug. Look in STDIO.H on the target for such nightmares.

The toupper Function

Purpose:

Converts a character in the range a-z to upper case A-Z.

Function Header:

```
int toupper(x)
int x;                          Character to convert.
```

Returns:

The character x, if no conversion was required.
The uppercase equivalent of x, if 'a' <= x <= 'z'.

Example: Convert an input line to all uppercase.

```
#include <stdio.h>
main()
{
char s[129];
int i;

     printf("Enter a line of text: ");
     gets(s);
     for(i=0;i<strlen(s);++i) s[i] = toupper(s[i]);
     printf("UPPERcase line:\n%s\n",s);
}
```

Notes:

- On some machines this function is a macro. Beware of side effects when porting code. If you write it, then try to do so without introducing possible side effects (like toupper(*p++)).
- Toupper() is an honest function, it is not a macro, and it has no side effects.
- Some macro implementations do not bother testing to see if the integer x to be converted is in the range a-z. The net result is an invisible bug. Look in STDIO.H for such problems.
- See also cvupper() for a function that converts an entire line to upper case with a single function call.

The ttyname Function

Purpose:

Get a device name string pointer from an open file descriptor.

Function Header:

```
char *ttyname(fd)
int  fd;                File descriptor for an open file, use
                        fd = fileno(fp) for a stream FILE *fp.
```

Returns:

(char *)p	Pointer to static memory containing the string for the device name.
(char *)0	Error, file not open or was not a valid device.

Example: Print the name of the device attached to stream fp.

```
#include <unix.h>
#include <stdio.h>
whatdevice(fp)
FILE *fp;
{
char *p,*ttyname();
     p = ttyname(fileno(fp));
     if(p == (char *)0) puts("Not a device");
     else puts(p);
}
```

Notes:

- Valid devices are con:, rdr:, pun:, lst:
- For Unix compatibility, assume this function returns only the name of the controlling terminal.
- The source code switches on device numbers as follows:

```
switch(fd) {
case   0:
case 252: return "CON:";
case 253: return "RDR:";
case 254: return "PUN:";
case 255: return "LST:";
default:  return 0;
}
```

The ungetc Function

Purpose:

Pushes a character back onto an open stream, ready to be used by the next call to getc(). Only a one-character push-back is allowed.

Function Header:

```
int ungetc(x,fp)
int x;                  Character to be pushed back.
FILE *fp;               Open stream pointer.
```

The value of x is ignored for a disk file, but it must be present in the argument list. For the console stream, the character x is placed in the console buffer, if possible (it could be full). An EOF character cannot be pushed back, and it is an error to attempt this feat.

Returns:

```
EOF       Push-back failed
x         Push-back worked (or x = EOF)
```

Example: Push a string back onto the console input buffer.

```
#include <stdio.h>
main()
{
char s[129],t[129];
int i;
     printf("Enter a string: ");
     gets(s);
     for(i=0;i<128;++i) {
          if(s[i]) ungetc(s[i],stdin); else break;
     }
     ungetc('\n',stdin);
     gets(t);
     printf("Here's t: %s\n",t);
}
```

Example: Read an input file until % is found, then print the line on which it occurs.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv;
{
char s[129];
FILE *fp,*fopen();
int i;
     if(argc <= 1) exit(0);
     if((fp = fopen(argv[1],"r")) == (FILE *)0) exit(0);
     while((i = getc(fp)) != EOF && i != '%') ;
     if(i == EOF) {
          printf("Flag %% not found\n"); exit(0);
     }
     else {
          printf("Found %%\n");
          ungetc(i,fp);
          fgets(s,128,fp);
          printf("%s",s);
     }
     fclose(fp);
}
```

Notes:

- Calls to fseek() or freopen() or seek() are unaffected by a call to ungetc(), because data in the file buffer is not changed.
- Calls to ungetc() for disk files may have the effect of moving the file pointer back to a previous buffer, in which case the action taken is to reload the file buffer from disk.
- The console limit is 136 characters, and you may attempt to push back an entire string to see how this works. Under re-direction this process will fail to have the desired result because the push-back for files does not fool with the data on disk.
- Crazy code that stuffs the file buffers with data that was not on disk is not acceptable. It will have to be re-written. Sane code will compile and run normally.
- Scanf() uses ungetc(). If your code mixes fseek, ungetc, scanf, then it may not be portable, even though it executes properly with this library.

The unix I/O and system Functions

Purpose:

Allow UNIX System III code to compile under C/80. Stubs are in the file UNIXIO.H, ready to edit into your program. A copy of all stubs is provided in CLIBU.REL, for convenience. See UNIXIO.H for more details.

Reference texts:

Basic Reference: Banahan & Rutter, The Unix Book, Wiley Press, New York (1983).

Unix Reference: The Unix Programmers Manual.

Function Headers:

```
=====UNIX SYSTEM CALLS======

     int acct(file)                    Return 0.
     char *file;

     unsigned alarm(seconds)           Return 0.
     unsigned second;

     char *cuserid(s)                  Return (char *)0.
     char *s;

     int dup(old)                      Return -1 for error.

     int dup2(old,new)                 Return -1 for error.

     int fork()                        Return -1 for error.

     #include <sys/types.h>
     #include <sys/stat.h>
     int fstat(fd,buf)                 Return -1 for error.
     int fd;
     struct stat *buf;

     #include <sys/types.h>
     #include <sys/timeb.h>
     void ftime(tp)                    Void return.
     struct timeb *tp;

     int getpid()                      Return 0.

     int getuid()                      Return 0.

     int getgid()                      Return 0.
```

```
int geteuid()                          Return 0.

int getegid()                          Return 0.

int ioctl(fd,request,argp)             Return -1 for error.
int fd,request;
struct sgtty *argp;

int getpgrp()                          Return 0.

int getppid()                          Return 0.

int kill(pid,sig)                      Return -1 for error.
int pid,sig;

int link(path1,path2)                  Return -1 for error.
char *path1,*path2;

int mknod(path,mode,dev)               Return -1 for error.
char *path;
int mode,dev;

int mount(spec,dir,rwflag)             Return -1 for error.
char *spec,*dir;
int rwflag;

int nice(incr)                         Return 0.
int incr;

void pause()                           Void return.

int pclose(fp)                         Return -1 for error.
FILE *fp;

int pipe(fd)                           Return -1 for error.
int fd[2];

FILE *popen(command,type)              Return 0.
char *command,*type;

void profil(buff,bufsize,offset,scale)         Void return.
char *buff;
int bufsize,offset,scale;

int ptrace(request,pid,addr,data)      Return -1 for error.
int request,pid,addr,data;

int putpwent(p,fp)                     Return -1 for error.
struct passwd *p;
FILE *fp;

int setpgrp()                          Return 0.
```

```
int setgid(gid)                         Return -1 for error.
int gid;

int setuid(uid)                         Return -1 for error.
int uid;

#include <sys/types.h>
#include <sys/stat.h>
int stat(name,buf)                      Return -1 for error.
char *name;
struct stat *buf;

void sync()                             Void return.

long time(tloc)                         Returns (long)0
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
long times(buffer)                      Return -1 for error.
struct tbuffer *buffer;

int stime(tp)                           Return -1 for error.
long *tp;

int umask(cmask)                        Return 0.
int cmask;

int umount(spec)                        Return -1 for error.
char *spec;

int uname(name)                         Return 0.
struct utsname *name;

char *userid(s)                         Return (char *)0
char *s;

int ustat(dev,buf)                      Return -1 for error.
int dev;
struct ustat *buf;

#include <sys/types.h>
int utime(path,times)                   Return -1 for error.
char *path;
struct utimbuf *times;  or  time_t timep[2];

int wait(stat_loc)                      Return -1 for error.
int *stat_loc;
```

=====UNIX MISCELLANY=====

```
#include <grp.h>
struct group *getgrent()            Return 0.

#include <grp.h>
struct group *getgrgid(gid)         Return 0.
int gid;

#include <grp.h>
struct group *getgrnam(name)        Return 0.
char *name;

int setgrent()                      Return 0.

int endgrent()                      Return 0.

char *getlogin()                    Return (char *)0.

int getpw(uid,buf)                  Return -1 for error.
int uid;
char *buf;

#include <pwd.h>
struct passwd *getpwuid(uid)        Return 0.
int uid;

#include <pwd.h>
struct passwd *getpwent()           Return 0.

#include <pwd.h>
struct passwd *getpwnam(name)       Return 0.
char *name;

int setpwent()                      Return 0.

int endpwent()                      Return 0.

int monitor(lowpc,highpc,buffer,bufsize,nfunc)    Return 0.
int (*lowpc)();
int (*highpc)();
short buffer[];
int bufsize;
int nfunc;

int sleep(seconds)                  Return 0.
unsigned seconds;
```

The unlink Function

Purpose:

Erase a disk file from the disk directory.

Function Header:

```
int unlink(filename)
char *filename;                    Existing file name.
```

Returns:

```
 0          No error.
-1          Error, operation failed.
```

Example: Delete a disk file.

```
#include <stdio.h>
main()
{
int unlink();
char s[129];
       printf("File name to erase: ");
       gets(s);
       if(unlink(s) == -1) puts("Erase failed");
}
```

Example: Delete a list of files.

```
#include <stdio.h>
main()
{
int unlink();
  if(unlink("A:*.COM") == -1) puts("Erase failed");
}
```

Notes:

- Unlink() is a Bell Labs standard function.
- The function accepts a wildcard argument. You may delete many files at once with a single call to unlink().

The utoa, utoax, utoao, utoab Functions

Purpose:

Convert 16-bit integer value to printable characters. String after conversion is between 5 and 17 bytes in length. See below.

Function Header:

```
int utoa(n,s)       Unsigned integer to decimal digits.
int n;              Integer to convert.
char s[6];          Storage area for the string.

int utoax(n,s)      Unsigned integer to hexadecimal digits.
int n;              Integer to convert.
char s[5];          Storage area for the string.

int utoao(n,s)      Unsigned integer to octal digits.
int n;              Integer to convert.
char s[7];          Storage area for the string.

int utoab(n,s)      Unsigned integer to binary digits.
int n;              Integer to convert.
char s[17];         Storage area for the string.
```

Returns:

The number of digits converted. Equals strlen(s).

Example: Print integer conversions.

```
#include <stdio.h>
main()
{
int i; char s[129];
     do {
          type("Enter number: ");
          gets(s);
          i = atoi(s);
          itoa(i,s); puts(s);
          utoa(i,s); puts(s);
          utoax(i,s); puts(s);
          utoao(i,s); puts(s);
          utoab(i,s); puts(s);
     } while(i);
}
```

Notes:

- All routines jump to the assembler interface $STRCV after loading the accumulator. Same routine as used by printf.

The waitz Function

Purpose:

Waits for a specified number of milliseconds.

Function Header:

```
VOID waitz(n)
int n;              Wait for 2n milliseconds. Uses the 2
                    millisecond memory clock of the Z90
                    and Z100 computers.
```

Returns:

Nothing. CPU is bound to a loop until timeout.

Example: Wait for time entered on the command line.

```
#include <stdio.h>
main(argc,argv) int argc; char **argv[];
{
int i,j;
  if(argc<=1) exit();
  j = atoi(argv[1]); if(j+j==0) exit();
  for(i=0;i<10;++i) {
  printf("Waiting for %dms..\n",j+j);
  timer(0); waitz(j); timer(1);
  }
}
```

Notes:

- If your machine does not have a memory-based timer, then look around for an 8253 timer chip or similar timing device.
- The timer interface is used because of accuracy. You can build a software timer that does essentially the same thing, but it will be sensitive to CPU speed.
- Works on the H89, Z90, Z100 with CP/M-85. Very little chance on other systems. See the source code for ideas on how to implement it on other computers.
- Attempts to test this function on machines that are not hardware equipped as cited above will result in a machine hang. The software loop is an infinite loop in case the timer does not change (unused location 11).

The wrDISK Function

Purpose:

Direct-disk write function for CP/M 2.2.

Function Header:

```
wrDISK(track,record,drive,buffer)

int track;              Physical track to write, 0 = first.
int record;             Record to write on track, 1 = first.
int drive;              Drive number. Use 0 for A:, 1 for B:, etc.
char buffer[128];       Location of data to write to disk.
```

Returns:

Buffer written on success.
Error code for a disk write is returned. Should be 0 for success.

Example: Write track 0 sector 1, the Zenith label sector.

```
#define libl CLIBX
#include <stdio.h>
main()
{
int i;
char buf[128];
#define TRK 0
#define SEC 1
#define DSK 0

     fillchr(buf,'\0',128);
     printf("wrDISK() = %d\n", i = wrDISK(TRK,SEC,DSK,buf));
     for(i=0;i<128;++i) printf("%02x\n",buf[i]&255);
}
```

Notes:

- Note that wrDISK() is coded in assembly language and is highly non-portable. The function wrDISK() can be written in terms of the function bios(). The latter is recommended for portability, especially to CP/M-68K and Unix. The portable source:

```
bios(12,buffer);                Under CP/M-68K the second
bios(9,drive);                  argument of bios() has a
bios(10,track);                 cast of (long).
bios(11,record);                Under CP/M-80 2.2, the cast
return bios(14,0);              is (int).
```

- Immediate writes to disk must use bios(14,1). This problem occurs with LRU buffering (Z100, for example). Normal writes use bios(14,0) with possibly delayed physical write.

The write Function for C/80

Purpose:

Writes a block of characters to the disk in binary mode. The amount written must be a multiple of 128. Adheres to the Software Toolworks standard for the write() function. Use UNIX.H to obtain the Bell Labs Unix standard.

Function Header:

```
#include <stdio.h>
unsigned write_(fp,buffer,count)
FILE *fp;                       Open stream pointer.
char *buffer;                   Buffer which contains data.
unsigned count;                 Number of bytes to write.
                                This number must be a multiple
                                of 128.
```

Returns:

The number of bytes actually written, which should equal the number count above. It is an error if write_() != count.

Example: Write 2048 bytes to a disk file.

```
#include <stdio.h>
put2048(fp,buf)
FILE *fp;
char buf[2048];
{
unsigned x,write_();
     x = write_(fp,buf,2048);
     if(x != 2048) puts("Write failure");
}
```

Notes:

- Streams with descriptors 252,253,254,255 are devices. Write_() under C/80 cannot use these device descriptors. Use putc() for safety.
- A write failure is usually due to end of media. Since the C/80 write_() function operates in binary mode only, there is no carriage return and linefeed translation. Nor does the function insert ctrl-Z at the end of the file.
- The write_() function under C/80 operates in quanta of 128 bytes. You cannot write 1 character. See UNIX.H and the Unix-style write() and fwrite() functions for an alternative.
- This write_ function uses CP/M function 34 with manual advance of the record number. The initial record number is saved in the system variable IOsect[fp].
- The symbol **write_** is the same label as **write** in CLIB.REL. Portable source code should use **write_** for the C/80 write(). Other systems will use read() and write() as in UNIX.H.
- Old C/80 code may be compiled without changes, because **write** is a global in CLIB.REL which has the correct meaning for such source code. New source code should use **write_**.
- A file opened for update "u" may be read to the end with the **read_()** function followed by rewind(fp) and access by the **write_()** function. Mixed calls of read_/write_/getc/putc will probably fail.

The write and writea Functions

Purpose:

Transfers bytes from main memory to a non-stream file. Move n bytes per call from a preset buffer. The transfer suffers from cr/lf translation if the file was opened in Ascii mode (the usual case).

Function Header:

```
int  writea(fd,buffer,n)
int  fd;              File descriptor, fd=fileno(fp) where
                      fp is the stream pointer.
char *buffer;         Pointer to base of memory storage which
                      contains the data to be written to disk.
int  n;               Number of bytes to transfer, 0...32767.
```

Returns:

```
m       The number of bytes actually transferred, an integer
        quantity 0...32767. Could be different from n.
-1      Error
```

WARNING: In UNIX.H appears the definition **#define write writea.**

Example: Write 1 byte to a file.

```
#include <unix.h>
#include <stdio.h>
main()
{
FILE *fp,*fopen();
int x;
     fp = fopen("TMP","w");
     x = 'A';
     write(fileno(fp),&x,1);
     fclose(fp);
}
```

Notes:

- Note that UNIX.H installs a new name for write(). You actually use writea(). Beware of the name conflict with C/80 write(). The primitive C/80 write() is used implicitly. All Unix re-direction is in force, because of the explicit use of putc().
- Useful for writing out Ascii data from the current file pointer.
- This function differs from the C/80 primitive write() in that quanta of 128 bytes are not required and in addition you can write to a device. Devices CON:, PUN:, LST: are recognized, but CR/LF translation occurs.

The writeb Function

Purpose:

Transfers bytes from main memory to a file opened by descriptor. Move n bytes per call from a preset buffer. The transfer is done in binary mode regardless of how the file was opened.

Function Header:

```
int  writeb(fd,buffer,n)
int  fd;              File descriptor, fd=fileno(fp) where
                      fp is the stream pointer.
char *buffer;         Pointer to base of memory storage which
                      contains the data to be written to disk.
int  n;               Number of bytes to transfer, 0...32767.
```

Returns:

```
 m      The number of bytes actually transferred, an integer
        quantity 0...32767. Could be different from n.
-1      Error
```

WARNING: In UNIX.H appears the definition

```
#define write writea
```

Example: Write a linefeed only to an output file opened in ASCII mode.

```
#include <unix.h>
#include <stdio.h>
main()
{
FILE *fp,*fopen();
int x;
     fp = fopen("TMP","w");
     x = '\n';
     writeb(fileno(fp),&x,1);
     fclose(fp);
}
```

Example: Write a linefeed only to the console.

```
#include <unix.h>
#include <stdio.h>
main()
{
int x;
     x = '\n';
     printf("Watch it happen");
     writeb(fileno(stdout),&x,1);
     printf("here");
}
```

Notes:

- The header file UNIX.H names write() as writea(), an ASCII access. Beware of the name conflict write() creates with the C/80 library.
- Useful for writing binary data from the current file pointer.
- The console device is treated specially so that a linefeed is not converted to CR/LF. For other devices like the printer, bytes are output just as they appear in the buffer without further translation.
- For disk files, the mode of the file is temporarily changed to binary and the bytes are output from the buffer to the stream buffer. Recall that under the present library, all disk files are stream files. However, the fill character used for the stream and the end-of-file mark are not affected.
- This function differs from the C/80 primitive write() in that quanta of 128 bytes are not required and in addition you can write to a device.